# A TWO-LAYER ENERGY-EFFICIENT WIRELESS SENSOR NETWORK FOR PRECISION AGRICULTURE APPLICATIONS

By

Osiris V. Ntarugera, B.S.

A Thesis Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Science

In

Electrical Engineering

University of Alaska Fairbanks

December 2018

APPROVED:

Dr. Dejan Raskovic, Committee Chair
Dr. Charles E. Mayer, Committee Member
Dr. Michael Hatfield, Committee Member
Dr. Charles E. Mayer, Chair
*Department of Electrical Engineering*
Dr. Doug J. Goering, Dean
*College of Engineering and Mines*
Dr. Michael Castellini*, Dean of the Graduate School*

## Abstract

The agriculture industry has benefited from the recent technological evolution; for example, farmers now use satellite images to monitor large fields. The use of technology in agriculture, generally referred to as Precision Agriculture, has attracted a lot of research interest from electrical engineers. One particular area of Precision Agriculture is the application of embedded systems in monitoring large crop fields. Sensor nodes are placed at various locations in the field where they measure different parameters, such as temperature and soil moisture. The collected measurements are sent to a central hub outside of the field where they can be further processed and displayed for the farmers to make appropriate decisions. From the farmers' perspective, this kind of wireless sensor network (WSN) is a cost-effective solution that allows them to gather accurate information about their crops in real time and significantly improve production. To scientists, it provides invaluable information that can help them improve farming processes or even develop new crop varieties. From the embedded systems stand-point however, such a network poses several challenges, mainly battery life and network lifetime. Battery life is a serious challenge because nodes are scattered in the field and it would be labor intensive and expensive to replace their batteries. It is important to keep nodes alive because dead nodes not only fail to collect data but they also fail to relay packets from other active nodes. Radio communication draws most of the node's battery in WSN, so most energy saving techniques revolve around careful management of the radio. In this study, we focus on routing protocols that maximize the lifetime of the network.

Most researchers have suggested various routing schemes to minimize battery consumption by finding the shortest path to a hub; however, when looking at the network as a whole, this approach may not be ideal. We present a lifetime-maximizing routing scheme that uses a cost function to distribute the traffic load among all nodes and to spare those with low remaining energy. The cost function being essential to our algorithm, we evaluate the impact of different types of cost function on the network lifetime. Lastly, we evaluate the impact of link quality in the cost function. Simulation results show that the power cost function has the best performance and that link quality can improve network lifetime.

Another major contribution of this research is the design of a test framework that can be used to evaluate other routing protocols. In order to evaluate our routing protocol, we created a WSN simulation in Castalia. The simulation and the routing protocol are highly parametric and with minor modifications, users can experiment with new protocols or variations of ours. Using our platform can

save users a lot of time and trouble, especially those unfamiliar with simulation tools, hence allowing them to focus their efforts on their protocol.

## Table of Contents

## List of Figures

Page

## List of Tables

Page

## List of Appendices

Page

# Chapter 1 Introduction

Wireless sensor networks (WSN) use small sensing devices that communicate wirelessly to monitor various physical parameters. These small devices, commonly referred to as sensor nodes or simply nodes, are scattered around the area of interest where they collect data using their sensors. In addition to collecting data, nodes communicate with each other. A typical sensor node has a processing unit, one or multiple sensors, a radio, and a power source. The growth of this new technology can be attributed to the development of small, low-power, inexpensive microprocessors such as Texas Instruments' MSP430 [1]. Sensors have also seen tremendous improvements both in size and cost, which allows users to deploy hundreds of them in a field at a reasonable cost. WSN have gained popularity in various applications due to several advantages they have over traditional means. First, WSN cuts down the wiring cost which has been estimated to US$ 130 – 650 per meter in industrial installations [2]. Other benefits include fast deployment, ability to work in dangerous, hazardous, or difficult access locations, and reduced maintenance complexity. These benefits have made WSN applicable to many areas, such as environmental monitoring (wild fires), military applications, infrastructure monitoring (bridges), and agriculture.

According to the US Department of Agriculture (USDA), agriculture, food, and related industries contributed $992 billion to U.S. gross domestic product (GDP) in 2015, a 5.5% share [3]. Like any other industry of that size, the farming industry has adopted several technologies, under what is commonly referred to as Precision Agriculture, to increase profitability, efficiency, safety, and to reduce the environmental impact [4]. For instance, farmers do not need to apply the same amount of fertilizers over the entire field but they can target specific areas needing more fertilizers. Such efficiency requires accurate knowledge about the state of the soil which can be achieved by using wireless sensor nodes to monitor various parameter in the farm. Those nodes are equipped with different types of sensors and an RF module. Sensors collect data on various parameters such as soil humidity, ambient temperature, and pH levels. Using their radio, sensors are able to wirelessly talk to each other and to send their data to a central hub where it is easily accessible. Various studies have been conducted to experiment with the use of WSN in agriculture, [5], [6], [7].

Using wireless sensors introduces several challenges that, if not addressed, can severely affect the performance of a network. The most common and probably challenging issue with wireless sensors is their limited energy capacity. Due to their size limitation, sensors are typically equipped with small

batteries with very limited capacity. Besides, it is often difficult and expensive to replace those batteries due to the harsh environment where the sensors are deployed. Battery management is particularly challenging for WSN applications in farming; first, because the network needs to live longer in order to monitor a certain crop and secondly, there are many obstacles in the farm that can interfere with radio communication and therefore reduce the link quality, leading to an increase in energy consumption. Hence, proper measures need to be taken to extend the battery lifetime of sensor nodes.

Since node batteries cannot be easily recharged or replaced, the only way to extend the node's lifetime is by minimizing the power consumption. Typical wireless sensor nodes, such as the popular Micaz [8], are mainly made of a microcontroller, an RF transceiver, and one or more sensors. Of all those components, the radio consumes the most power and therefore most of power saving techniques revolves around managing the radio resource carefully. In this study, we are particularly interested in saving power through a smart routing protocol.

Routing for wireless sensor networks has been extensively studied and there are many protocols that have been proposed by the research community, as presented in [9]. Although most of routing protocols are designed to achieve the same goal - finding the most efficient path to the sink, they widely differ in their approach. In this study, we focus on *Lifetime-maximizing energy-aware routing techniques*. These protocols achieve the best network lifetime performance by sharing the load among nodes while sparing nodes on the brink of exhaustion. Load-balanced protocols such as [10] and [11] use a cost function to determine which route to use. Although the cost is often a function of the remaining energy and the initial energy, different types of functions are also used. We propose an exponential cost function because of the flexibility and smoothness of the exponential function. In addition to energy, our route cost function takes into account the link quality which is very unpredictable in harsh environments like a farm.

For the purpose of this study, we define network lifetime in terms of active sectors instead of active nodes. The idea here is to monitor how many nodes are active in any given area of the covered field. When looking at coverage over the entire network, it is possible to have an uncovered area while the network as a whole looks healthy. Therefore, our network is divided into sectors and a sector is considered active when it has at least a certain number of working nodes. That number is selected by the user and it depends on how much coverage is needed. We see this concept as one of the important contributions of this thesis.

Our network is randomly deployed and we assume that nodes use an equal amount of power to communicate among each other. The network is also densely populated to create some redundancy, which allows the network to stay alive despite losing some nodes.

The rest of this document is organized as follows: chapter 2 discusses relevant work in the research community, both about routing and precision farming; chapter 3 describes the simulation tool that was used in this study; chapter 4 talks about the network architecture and the routing protocol; chapter 5 presents the simulation results and their discussion; and finally, chapter 6 concludes and discusses future improvements.

# Chapter 2 Literature Review

## 2.1 Precision Agriculture

Wireless sensor networks are making their presence felt in many aspects of our lives. Among other fields, they have found their way in agriculture under what is commonly called Precision Agriculture. The benefits of precision farming come in twofold: profitability for users and ecological and environmental protection for the public [12]. Having information about the state of the farm, farmers are able to make informed and profitable decisions, such as applying fertilizers and other agrochemicals based on spatial and temporal variability. They can evaluate the input cost vs return of a given field. All of this allows them to increase the yield while saving money. Reducing the amount of water and fertilizers also reduced the environmental impact of farming.

### 2.1.1 PA history and current trends

Precision agriculture (PA) is defined as a management practice capable of improving benefits by utilizing more precise information about agriculture resources [13]. PA research started in the late 1980s, first in developed countries in North America, Australia, and Western Europe [2], but it has spread to other countries as well. In mid 1990s, technological advances in PA attracted Chinese agriculture engineers [12]. In 1998 a nationwide survey by USDA revealed that 4% of all farms used one or more PA technologies [14]. A similar survey in Arkansas indicated that 20% of Arkansas farmers had adopted PA [15]. According to Global Market Insights, the precision farming market was estimated to be over 3.4 billion USD in 2016 and it is expected to grow at a 14% compound annual growth rate (CAGR) [16]. Sensors are predicted to dominate the hardware market, covering 19% of the market by 2024. With such potential, this market has seen a number of companies develop, namely: AGCO Corporation [17], Agribotix providing drone-based solutions [18], AgSense LLC specializing in remote-controlled irrigation systems [19], etc.

### 2.1.2 Wireless sensor networks in PA

There are various technologies currently employed by farmers to monitor their fields and to manage their resources more efficiently, such as water and fertilizers. The following are five major technologies employed in PA: Geographical Information Systems (GIS), Global Positioning Systems (GPS), wireless sensors, Variable Rate Technology (VRT), and Yield Monitoring (YM) [13]. Wireless sensors have been growing in popularity mainly because of their declining cost; Crossbow Technology Inc. predicted their price to drop by 50% between 2005 and 2009 [2] and that trend still continues [20]. Furthermore, WSNs

applications in agriculture are divided into five categories: environmental monitoring, precision agriculture, machine and process control – machine-to-machine (M2M) communication, facility automation, and traceability systems. PA sensors are classified in five categories: yield sensor, field sensors, soil sensors, crop sensors, and anomaly sensors [12]. Using sensors, farmers are able to collect data from their field in real-time – a process that would otherwise take days or even weeks using a traditional way of collecting soil sample and sending them to a lab [21]. Aqeel-ur-Rehman et al. list and compare sensors and technologies used in agriculture; they mention sensor nodes such as MICAz and MICA2, and communication standards such as ZigBee and Bluetooth that are commonly used in precision agriculture [22].

### 2.1.3 Examples of WSNs Applications in PA

WSNs can be applied in different aspects of farming, such as irrigation, fertilization, pest control, and more [22]. For example, image sensors, such as RGB and hyperspectral, were used for early detection of plant diseases [23]. In 2003, Discovery Channel reported a WSN application in a vineyard in British Columbia, Canada [2]. In a successful experimental project, wireless sensor nodes were used to monitor the soil moisture and the air temperature in a cabbage farm in Southern Spain [7]. The monitoring system remained active and accurate for the entire growing season for cabbage, 10 weeks. In another experimental study, researchers from Delft University deployed 109 sensor nodes to monitor humidity and temperature in a potato field [5]. Another study presents a SN monitoring system capable of detecting and identifying intruders and alerting the farmers upon detection [6]. Motorola Labs developed neuRFon, a low-cost, low power, and self-organizing sensor network that can be used to sense agriculture and environmental parameters [24]. Damas et al. proposed a distributed, remotely controlled, automatic irrigation system for a 1500 ha area in Spain [25].

### 2.1.4 Challenges faced by WSNs in PA

Using WSNs faces some challenges. Lack of complete standardization, overwhelming amounts of data generated, and power supply are some of the obstacles that WSNs in PA need to overcome [2]. As we find in most of WSNs applications, the main challenge that arise from designing and implementing WSN in agriculture: Energy consumption is the main challenge because nodes need to preserve energy for a long time to be able to assume their responsibility; other issues include fault tolerance or network immunity, node size and housing [22]. The problem of power limitations was also mentioned by Perkins et al. in their neuRFon sensor project [24]. Despite their efforts to minimize current consumption, their test bed, powered by two 1.35 V Silver Oxide cell batteries from Energizer, was not able to last one full

day. Despite the challenges, the WSN market has experienced growth mainly due to the decline of sensor costs. According to a report from the Atlas, the cost of IoT sensors has been falling consistently over the last couple of decades – the average cost of IoT sensors is expected to drop to US $0.38 in 2020, which is about 30% of what it was in 2004, US $1.30 [20].

The researchers from Delft University shared the lessons they learned from their experimental WSN project application in agriculture [5]. In this pilot project, originally initiated to obtain experience in WSN deployment for PA, 109 sensor nodes were deployed in a potato field to detect the conditions for Phytophtora, a fungal disease that spreads fast and destroys potato plants. Although their project ran into more issues than had been anticipated and failed its original mission, it provided a wealth of knowledge about challenges of deploying a WSN in a farm. Among many challenges encountered, they mention the failure of the MinRoute routing protocol that was used in combination with TMAC as part of the network stack. For example, MinRoute introduced long paths for nodes that were only one hop away from the gateway. Normally, MinRoute selects a parent node from a set of neighbors based on their statistics. Nodes with poor statistics are removed from the neighbor list. Consequently, in this experiment, most of nodes did not have the gateway in their neighbor list. Also, many packets were dropped due to difference between MinRoute's neighbor list and TMAC's list. This particular problem shows the need for a specialized routing for precision agriculture.

### 2.2 Routing Protocols

Routing has been widely studied in the research community from different angles. Although routing protocols take various approaches, they are often trying to solve the same problem: reduce the amount energy it takes to send packets across the network which extends the lifetime of sensor nodes and the network as a whole. In this section we briefly go over several routing protocols that have been suggested in the research community; a more extensive list of routing protocols can be found in [9].

#### 2.2.1 Energy efficiency routing

One of the most popular routing algorithms in the research community is LEACH (Low-Energy Adaptive Clustering Hierarchy), thoroughly described in [26]. The paper starts by describing two approaches commonly used to collect data from sensors at the base station. The first one is the direct communication where nodes directly talk to the base station, adjusting their communication range as necessary. Despite being the least complex computationally, this approach performs poorly in terms of energy conservation. Nodes that are far from the base station need to transmit at a higher power to be

heard so they die faster leaving a gap in the network. The second approach is called minimum-energy (MTE) where nodes use intermediate nodes to reach the base station. This approach solves the problem of energy-draining long-range communication but it suffers because nodes that are closer to the base station handle a lot of traffic from other nodes. Contrary to the first approach, in MTE, nodes closer to the base station die faster and still leave a gap in the network. In the proposed approach, LEACH, nodes form clusters randomly and elect temporary cluster heads. Nodes send data to the cluster head which then relays the information to the base station. In LEACH, nodes alternate the cluster head responsibilities to extend the overall network lifetime. If only certain nodes served as cluster heads it is clear that they would die faster because they handle more traffic. But in this approach, nodes periodically re-elect cluster head and in each round only those nodes that have not been cluster-heads are chosen from. Simulation results show that LEACH achieves between 7 and 8 times more energy reduction compared to the direct communication approach and between 4 and 8 times compared to MTE. Additionally, it takes about 8 times longer for the first node to exhaust its battery and approximately 3 times longer for the last node to die as it does for the two other approaches [26]. Last but not least, when the nodes start to run out of power and die, they do so in a random fashion which does not affect the network coverage as much as nodes in one area all dying. LEACH has truly been a state-of-the-art innovation compared to the older approaches and that explains its explosive success among the research community. Future improvements of the LEACH algorithm take into account the node's remaining energy in the cluster-head election process. The original approach assumes the nodes to use energy fairly equally but in a real network, that is often not the case.

Base Station Controlled Dynamic Clustering Protocol (BCDCP) is another clustering technique to save energy [27]. Unlike in LEACH, clustering decisions in BCDCP are made by the base station which has more energy and more knowledge on the network status, an approach also used in LEACH-C, an improved version of LEACH [28]. The key improvements in BCDCP are the formation of balanced clusters, uniform placement of cluster heads, and using cluster-head-to-cluster-head data routing to the base station. From simulation results, BCDCP reduces the energy consumption by 30% and 40% compared to LEACH-C and LEACH respectively. BCDCP respectively improves network lifetime by 100%, 30%, and 5% over LEACH, LEACH-C, and PEGASIS – another cluster-based protocol [29]. BCDCP also offers a better packet delivery rate than the other three cluster-based routing protocols.

Farazandeh et al. propose the Hybrid Energy-Efficient routing (HEE), a combination of direction transmission (DT) and minimum transmission energy (MTE) [30]. DT and MTE are two of the simplest

routing techniques in WSN. In DT nodes communicate directly with the base station while in MTE nodes use intermediate neighbors to reach the base station. As previously mentioned, both DT and MTE have their own disadvantages [26]. DT performs better in small networks while MTE does better in larger networks. Nodes in HEE compute the energy to transmit using DT and MTE and they select the most energy effective. Simulations results show that HEE outperforms both DT and MTE in energy efficiency.

Singh et al. recognize the limitations in common metric-based routing such shortest-hop metric to conserve the energy [31]. They argue that using power-aware metrics results in energy consumption efficiency. They suggest a metric to minimize $c_{j=} \sum_{i=1}^{k} f_i(x_i)$, where $c_j$ denotes the cost to send a packet j from node i to node k and $f_i(x_i)$ a cost function of node i based on its expended energy thus far. Using simulation, they compare the shortest-hop routing with their shortest-cost routing, using a linear and a quadratic form of $f_i(x_i)$. The shortest-cost routing results in 5-15% improvement in terms of cost/packet and 5-10% in terms of reduction of maximum node cost. The authors also noted that improvements are even better for larger networks because they have more routes to choose from.

### 2.2.2 Load balancing routing

Shah and Rabaey propose an energy aware routing protocol for wireless sensor networks [32]. The paper explores the shortcomings of previous routing schemes that find and use the same optimal path to the destination. The problem with that approach is that it drains the nodes on the optimal path much faster than the rest of the network. The goal of this routing is to improve network survivability, i.e. maintaining network connectivity for as long as possible. In order to achieve that, this algorithm selects and alternatively uses a number of paths in a probabilistic fashion. More specifically, each node saves a number of routes to use to send data and each route is assigned a probability depending on its cost, costly routes having lower probability. The cost is determined using a cost metric on a link which is determined based on the energy to transmit on that link and the remaining energy of the sender. Nodes calculate their average cost as the sum of all of its neighbors (potential routes) weighted by their probability. This cost is broadcast to neighbors so they can update their tables. When a node needs to send data, it randomly chooses one route from its table based on its pre-calculated probability. This process is followed until the packet reaches its destination. This algorithm reduces the average energy consumption per node by 21.5 % mainly due to low overhead but it also reduces the energy difference between nodes. In addition to that, simulation results show that energy aware routing has a 44% improvement over diffusion in terms of lifetime defined as the time for first node to die.

Distributed Energy Balanced Routing (DEBR) is another load balancing routing protocol [33]. Unlike most routing protocols that make the assumption that the rate of event generation is uniform over the entire network this paper considers the scenario where events (data) are randomly generated in some parts of the network and it proposes a robust routing, DEBR, to efficiently handle such variation. DEBR uses a localized route decision mechanism where nodes select the next hop based on the remaining energy and the cost of transmission. Simulation results show that DEBR outperforms direct communication (DC), minimum transmission energy (MTE), and self-organized routing (SOR) [34] both in network lifetime and energy balancing. Furthermore, DEBR superiority is maintained for both random and repeated event generation which proves its robustness.

There are other load balancing routing protocols that use a cost function. DCFR, for example, is another cost-based routing that takes into account the energy consumption rate in addition to the available energy [11]. Load balancing routing like DEBR [33] use the nodal remaining energy to determine the cost of a given route. That approach however does not address the issue of a short route with low remaining energy and low total cost; this route would be favored over long healthier routes due to its relatively low cost. Using a combination of exponential and sine functions, DCFR establishes a cost function where small changes in the nodal remaining energy result in large changes in the cost function value. Through simulation, the authors prove that DCFR gives the network longer lifetime than direct communication (DC) [26], minimum transmission energy (MTE) [35] and DEBR.

Although these other routing protocols can save energy, they are not suitable for applications in agriculture due to the unique challenges of the environment and the need for the network to stay alive for the entire growing season. Besides, in this study we measure network lifetime in sector coverage, which provides more meaningful insight on the network coverage. Lastly, the vast majority of studies we encountered use a simplistic energy model that only considers the radio energy. As we will show later, a significant amount of the energy is consumed during the sleep mode and therefore we needed a more advanced energy model.

# Chapter 3 Castalia – WSN SIMULATOR

## 3.1 Why use a simulator?

Simulation tools are very useful in research because they allow researchers to recreate their experiment multiple times with same or different parameters and to obtain results faster and at the lowest cost. Simulation is particularly useful in wireless sensor network because the cost of deploying a network with hundreds of sensor nodes is very high and unaffordable for many researchers. Hence, our research in WSN takes advantage of the various simulation tools available on the internet. The goal of the research being to study the behavior of wireless sensors spread across an agriculture field which can easily cover hundreds or thousands of acres, the use of a simulation tool seems both beneficial and inevitable. Instead of building a network of more than a hundred nodes, each with hardware components such as the MCU, radio, sensors, etc., the network will be virtually implemented in a software simulation tool. That being said, it is essential to find a good simulation tool because it will affect both the quality of results and the amount of work to get meaningful results.

## 3.2 Choosing a Simulation Tool for Wireless Sensor Network

Choosing the right simulation tool turned out to be a more difficult challenge than anticipated. There are quite a lot of simulation tools for WSN available on the internet, more than a dozen. Sundani et al. survey and compare fourteen WSN simulation tools [36], and that is not an exhaustive list of all the available tools. Most of these simulation tools were built by research groups from different universities to be used in their own studies and then published on the Internet for public usage. They are written in different languages and therefore they look and work differently even though they are meant to accomplish the same general goal, simulating sensor networks. Additionally, because those tools are free software targeting a small community of researchers interested in WSN, they often do not have a friendly user interface and have very little documentation available. All of those factors made it extremely difficult to choose one simulation tool for this research. An ideal simulation tool for this research would have the following features:

- Allow a large number of nodes,
- Allow custom routing protocols,
- Power management feature,
- A friendly GUI allowing the user to change parameters and to display results.

There are several papers written about the various simulators [37] [38]; they generally describe the tool functionality, the developers of the tool, the language(s) the tool was written in, and ultimately they share the pros and cons of each tool. NS-2, for example, is one of the most popular tools in the research community [38] and it supports many protocols for different communication layers [37]. But NS-2 is also known to be complex and difficult to use mainly due to its use of the not very common Tool Command Language [39]. Unfortunately, it looked like the only way to determine with certainty the usefulness of each simulator was to actually use it for some time. That, however, is very impractical considering that it takes several days to a couple of weeks to learn how to use a new tool. In the following section, we present a few simulation tools that were considered for this research with more details.

### 3.2.1 Existing WSN Simulators

Castalia was the first simulation tool that was considered for this research. It is based on the Omnet++ platform which is a larger and more powerful network simulator [39]. Castalia was developed by a group of researchers at the National ICT Australia [36]. It was originally selected because of its highly parametric structure, its realistic wireless channel, and radio [40]. A node in Castalia comprises of several modules representing different network layers: application, routing, MAC, and more [41]. Those modules intercommunicate to ensure proper functionality of the node and the user has the option to define each module either from the built-in samples or by building a custom one. This is a particularly important feature because the goal of this research is to study a custom wireless sensor network. For that reason, some of the modules will need to be built from scratch. Following the rich Castalia's user's manual [41], any module can be written as a C++ class which is saved to the appropriate folder together with other similar modules. Then the user creates a configuration file where the network simulation is created by selecting modules as well as setting important simulation parameters, such as number of nodes and simulation time. The simulation is run from the Linux terminal since Castalia runs on Linux. The results can be visualized in the Linux command window or saved to a file for graphing. The main difficulties of using Castalia for the first time are the lack of a user interface and the complexity of creating modules. Without any GUI, interaction with the simulation tool, whether it is setting up parameters or visualizing results, is fairly challenging and inconvenient especially for new users. Creating modules is also time consuming because one has to design the module, write the code for it, and potentially debug it.

Another simulation tool that was considered is NS-2 which stands for Network Simulator 2. According to [39], NS-2 is a discrete event simulator built in C++ and OTCl (Object Oriented extension of Tool

Command Language). It is probably the most popular simulation tool for computer networks due to its extensibility [36]. In addition to its modular approach, which makes it very extensible, NS-2 has a lot of support and there are many protocols already implemented. However, it is important to mention that NS-2 can be used for both wired and wireless networks [39]. Just like most simulation tools, NS-2 has some limitations and drawbacks. According to [36] "NS-2 has a long learning curve and requires advanced skills to create meaningful simulations" and allows limited customization for things like packet formats, energy models, etc.

A number of other simulation tools were reviewed but dropped mainly due to the lack of support and documentation on the web. As mentioned earlier, the majority of tools were developed by researchers for their own study and development was discontinued and therefore those tools are lacking in functionality and support.

### 3.2.2 Important considerations while selecting a simulator

Although it is very difficult to say which simulation tool is the best for wireless sensor networks, here are a few things to consider in the process of choosing the right tool:

- The user needs to determine what they want, i.e., which aspect of the network they are interested in most. That is important because most of the tools have an advantage over the others in at least one aspect of the network simulation.
- The user needs to consider simulators that are actively supported and have good documentation. One way to find that out is to look up the last time when the simulator was updated and how many people talk about that tool. Some tools have discussion forums on the popular platforms such as Google Groups or Stack Overflow. Online support becomes particularly important when the user runs into errors while building a simulation. Moreover, reviewing discussions about a simulator can give the user an idea on what challenges to expect and the overall user experience.
- Lastly, the user has to take into account his or her programming skills. Simulators are built in different languages and more often than not, they require the user to write some code in order to create the desired simulations. It will save the user a lot of time and trouble if they choose a simulation tool that uses a language in which they are proficient.

Ultimately, getting one's hands dirty and running simulations on a tool is the only way to know with certainty if it is the right one.

After several weeks of searching the web and keeping those aforementioned criteria in mind, Castalia was determined to be the best fit for this project. Our research will be mainly focusing on routing which is supported in Castalia. Secondly, Castalia has a well-written and easily accessible user's manual and one of the most active discussion forums. In fact, earlier this year, a new version, Castalia 3.3, was released with bug fixes and some upgrades. The most common issues that occur while using Castalia are addressed on the discussion forum [42] which was created by its developers, active members themselves. Lastly, Castalia modules are written in C++ with which I personally have had good experience with. Castalia is an open source and flexible tool allowing the user to change most of the parameters and to design custom network simulations.

### 3.3 Castalia WSN Simulator

Castalia is a modular and highly parametric simulator for wireless sensor networks (WSN) and body area networks (BAN) [41]. It is based on the Omnet++, an extensible, modular, and component based framework for building network simulators [43]. Besides its modularity and flexibility that allows the user to build virtually any configuration, Castalia boasts one of the most realistic implementations of the wireless channel and radio model as we will see in section 3.4. The Castalia software package comes loaded with implementations of some of the most popular protocols for MAC and the radio module.

Figure 1 below shows a basic structure of the network in Castalia. Networks in Castalia have three main parts: a set of nodes, the wireless channel, and the physical process. Nodes represent physical sensor nodes and they perform exactly the same tasks. The wireless channel is the medium that allows nodes to communicate with each other since they are not directly connected as Figure 1 shows. The physical process represents the physical environment being monitored by the sensors. Although all three components are open source, research has mainly focused on the node aspect of the network since it has the most interesting layers: application, routing, MAC, and radio.

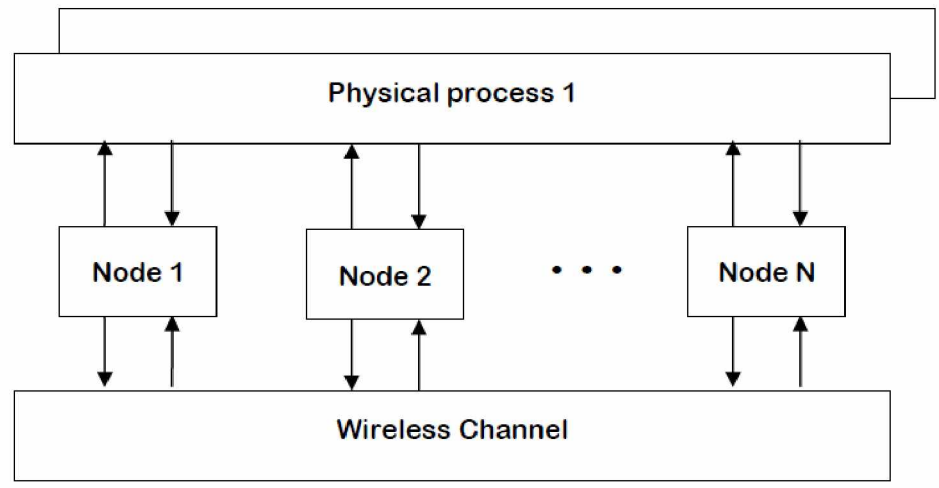



*Figure 1: Network overview in Castalia*

As Figure 2 shows, a node in Castalia is made up of several modules that work together to simulate the functionality of real sensor nodes. Each module is a separate entity performing specific tasks and able to communicate with other modules. The MAC module, for example, is responsible for medium access and it communicates directly with the routing and the radio module. Each module is implemented as a C++ object based on the parent class provided with the Castalia. Communication between modules occurs through messages, another feature that Castalia inherited from Omnet++ [41].

When building a new simulation, the user has an option to build nodes using the built-in modules or to design custom ones. In our research, we designed a custom application and we used built-in code for other modules. More details on writing modules and building simulations can be found in the user's manual [41].

Castalia runs in Linux and any interaction with the tool occurs mainly through Linux shell commands. This is probably the biggest disadvantage of Castalia because using command shell is not very intuitive and can be challenging for new users. Fortunately, Castalia has a very rich and well written user's manual that clearly lists all the steps needed to run a simulation. Simulation results are also printed in the command shell and the user is able to select which output they want to see. For visualization, Castalia provides some scripts to generate plots such as histograms but in our study, we exported data into Matlab for more advanced visualization. More details are provided in the results section.

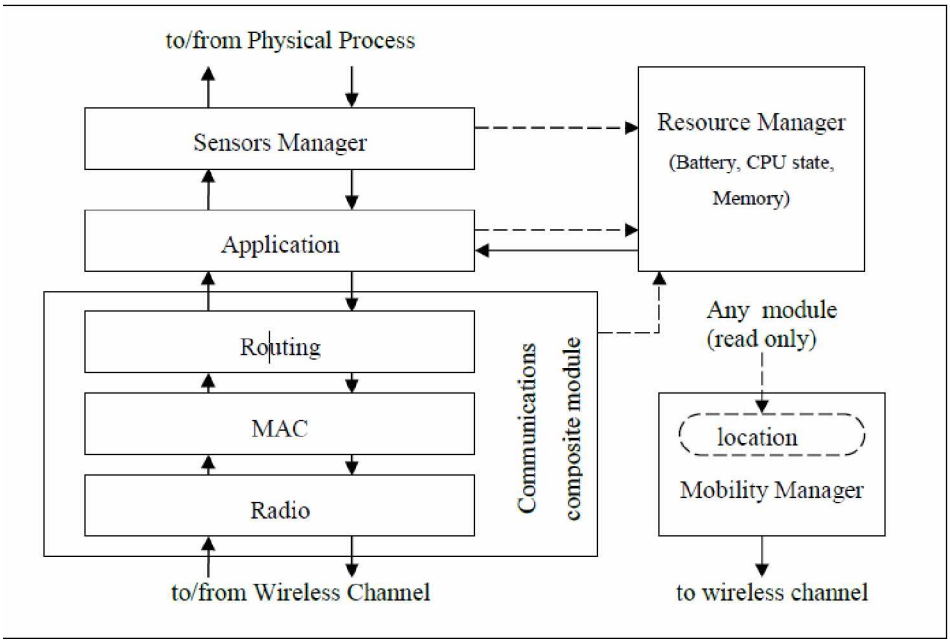



*Figure 2: Internal structure of a node in Castalia*

### 3.4 Castalia Validation for Model Correctness

One of the key aspects to be considered before conducting experiments in a simulation tool as mentioned in [37] is the correctness of the model. That means making sure that the different parts of the network simulators are designed according to correct and tested models. Inaccurate models generate incorrect results that do not reflect the behavior of real WSN, which can lead to misleading conclusions. It is therefore important to validate the simulation tool's model correctness before using it. Castalia was tested for model correctness. In its user's manual [41], Castalia is said to have one of the most realistic radio and wireless channel models so we decided to put the two modules to the test.

#### 3.4.1 Wireless Channel Model Test

First, we tested the path-loss modeling of the wireless channel. Castalia uses the Lognormal path-loss shadowing model [44], one of the most widely accepted models for radio propagation in short range wireless communication. As shown in Figure 3 below, the measured received signal power at various distances is consistent with the lognormal curve. The received signal strength data points are slightly scattered around the curve instead of lying on it perfectly due to the randomness of the channel, which is represented in equation (1) from [45], as a Gaussian random variable ($X_\sigma$) with a σ standard deviation. The results in Figure 3 were obtained with a standard deviation of 3.

$$PL(d) = PL(d_0) + 10 * \eta * \log\left(\frac{d}{d_0}\right) + X_\sigma \,. \quad (1)$$

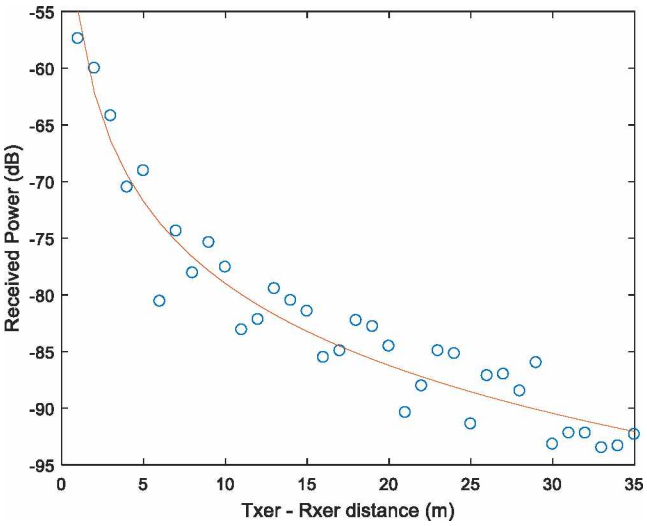



*Figure 3: Lognormal path-loss test results from Castalia's wireless channel*

### 3.4.2 Packet Reception Rate Test

The second test was to evaluate the correctness of the radio model. This test was inspired by section 5.2.3 *Radio Reception Model* in [45] which talks about the relationship between the packet reception rate (PRR) of a radio and the received signal-to-noise ratio (SNR). That relationship has been thoroughly studied in [44] and the PRR can be obtain as a function of the SNR, see Equation 2:

$$PRR = (1 - \frac{1}{2} * exp^{\left(-\frac{SNR}{2} * \frac{1}{0.64}\right)})^{8L}. \quad (2)$$

To evaluate the relationship between the PRR and SNR, we created a simulation where 100 packets are sent to a receiver located at various distances from the transmitter and we recorded both the received signal strength and the number of received packets. Figure 4 shows the PRR plotted versus the Received Signal Strength Indicator (RSSI). As expected, the PRR exponentially drops as the received power nears the sensitivity threshold of the radio, -95 dBm in this case. The simulation results do not perfectly match the analytical model, see Equation 2, but that is not surprising because the exact function of the PRR depends on the radio parameters, more specifically the encoding scheme and the modulation type [45] [44]. Equation 2 was developed for the Mica 2 mote radio which uses the NRZ encoding and the non-coherent FSK modulation [44]. Castalia Simulator uses the CC2420 [46] which has different characteristics [41].

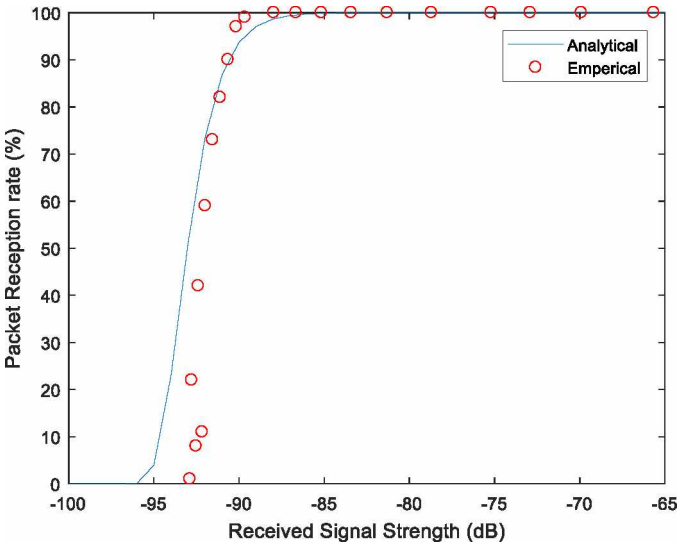



*Figure 4: Packet reception (PRR) ratio as a function of Received power in Castalia*

After those two simulation tests for the correctness of the radio and the wireless channel models, we concluded that the Castalia Simulation Tool meets our model correctness standards and is good for our study.

# Chapter 4 Network Architecture

## 4.1 Two-tier Deployment

Our network is made of 600 wireless sensor nodes deployed over an area of 200 × 200 m. The sensors were deployed in two layers, one primary layer of nine sink nodes deployed in a 3 × 3 grid and a secondary layer of data collecting nodes randomly deployed over the entire area, see Figure 5 below. The network area is divided into nine equally-sized sectors and each sector corresponds to one sink node (primary layer) located at the center. After deployment, each node in the secondary layer determines the sector where it is located, as explained in the following section.

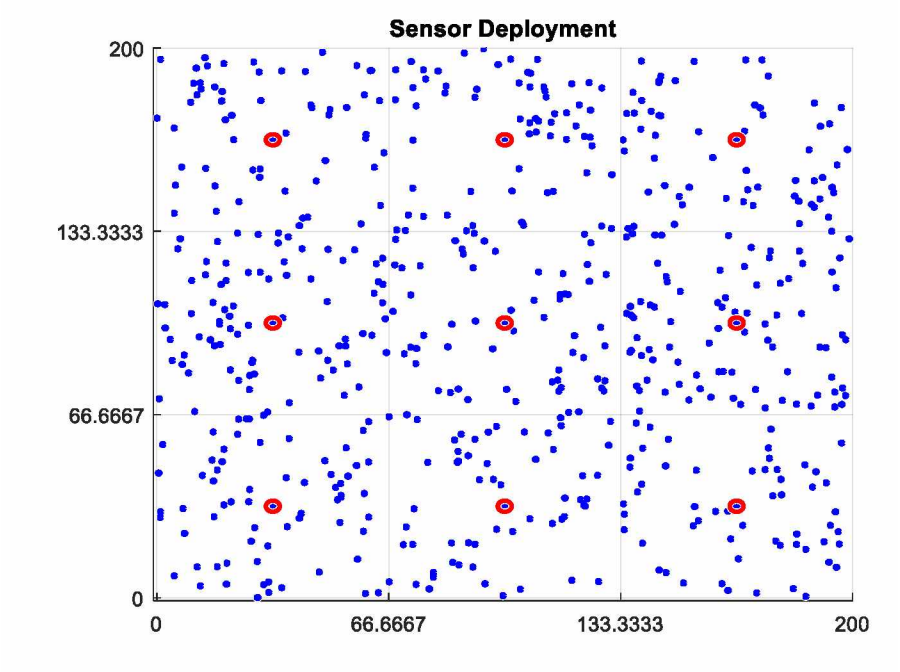



*Figure 5: 600 nodes deployed in 2-layer network over 200x200 m area*

## 4.2 Sector allocation

In this step, nodes determine which of the nine sectors they are located in. This step is particularly important because the accuracy of experimental results depend on the accuracy of this step. As we will see in the results section, the performance of our network is evaluated in terms of active sectors and sector activity is determined based on the number of active nodes in that sector. After deployment, each node needs to be assigned to a sector in order to determine how many nodes are in each sector and that is the goal of this step.

The sector allocation algorithm has two main steps:

**Step 1**: Each hub transmits a series of packets at 0 dBm and each node that receives a packet uses the source of the packet to determine where it is located, i.e., which sector it lays in. When a node receives multiple packets, it compares the RSSI and selects the one with the highest strength. Usually, the higher the RSSI the closer the transmitter [44]. Each node that received a packet from a hub broadcasts its location to other nodes for step 2.

**Step 2:** The algorithm runs another step where a node uses the location of its neighbors, from step 1, to determine or refine its own location. This step targets nodes that were left unconnected or connected to the wrong hub after Step 1. For example, in Figure 6 below node C was unable to determine its location in Step 1 but since it has three neighbors in Sector 1 and only one in Sector 2, it chooses Sector 1 as its own location. This approach allows all the nodes that were unable to hear from hubs to find their location. It also allows nodes to verify their location against their neighbors' for increased accuracy.

In some scenarios such as the one in Figure 7, the algorithm would incorrectly assign node C to sector 1 although it is actually located in sector 2. This scenario and many other scenarios that may occur calls for an improved algorithm and further studies can be done to improve this sector location algorithm using more information such as the RSSI. For the purpose of this study, this algorithm was deemed adequate because it addresses the more common scenarios. Besides, it would be very difficult and time consuming to write an algorithm that works for all the possible unusual scenarios that can happen in a random deployment. As our simulations showed, these unusual cases are very rare.

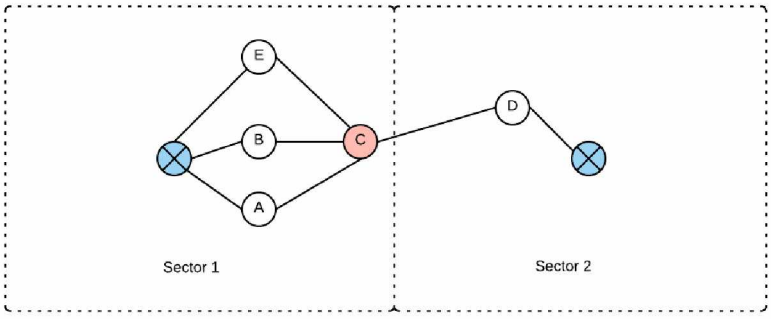



*Figure 6: Step 2 - finding location based on neighbors*

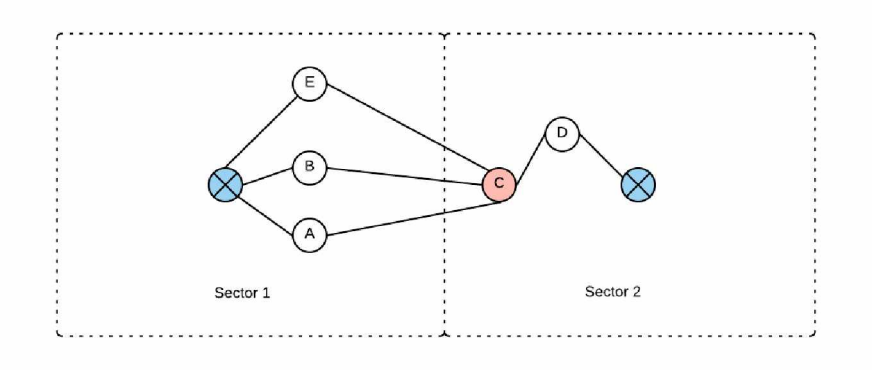



*Figure 7: Sector location potential flaw scenario*

Figure 8 below shows the results of the sector allocation algorithm. The majority of nodes were able to find the correct sector they are located in. A few nodes however, were incorrectly assigned to a sector next to the one they are actually in. This imperfection of the algorithm results from two factors; first nodes that are near the sector boundaries are likely to hear an equal strength signal from two hubs with the difference being the random noise in the signal. Secondly, the nodes refine their location based on their neighbors. So if a node in sector 1 is in proximity with several nodes from sector 2, it will choose sector 2 as its sector location. This problem is illustrated in Figure 7.

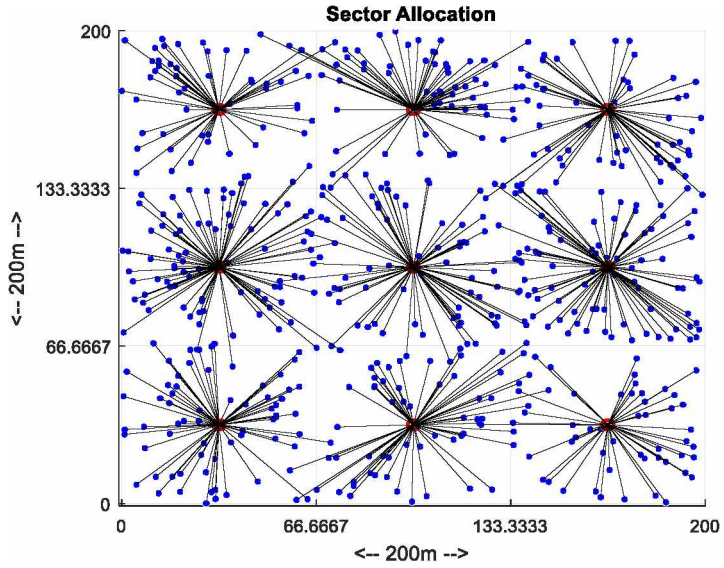



*Figure 8: Nodes connected to their respective sector-hub after the sector allocation algorithm*

### 4.3 Node density

One important factor to consider in a random deployment WSN is the number of nodes to be deployed or the density of the network. A dense network allows nodes to have multiple routes to the hubs and as we will show in the results section, that improves the network lifetime performance. High density does not come without a cost though because a dense network often suffers a lot of collisions which leads to loss of data and battery drainage.

In order to determine the ideal density for our network, we did the following experimental study. In the study, we simulated the network with different densities and we observed the key network qualities, such as connectivity and collisions.

The results were combined in order to comparatively study the effects of the network density on the network quality, as shown in Figure 9 and Figure 10 below. Figure 9 shows two graphs, the average number of neighbors per node and the normalized average number of collisions per node. The most obvious observation is that increasing the number of nodes increases the number of neighbors or the connectivity. And as predicted, raising the network density increases the number of collisions or packet failures, which can lead to poor battery usage. Figure 10 shows that as the density increases, there is a higher percentage of nodes with at least two neighbors.

Another observation from Figure 9, perhaps less obvious, is that from 120 nodes up, the connectivity grows at a slow rate while the average number of collisions grows at a much higher rate. For instance, the average number of neighbors for the 250-node network is only 12% higher than in the 150-node network; collision packets, on the other hand, rise by a staggering 128%. That goes to say that we cannot increase the number of nodes indefinitely for the sake of connectivity. In other words, at some point the collisions side-effects of a highly dense network outweighs its connectivity benefits.

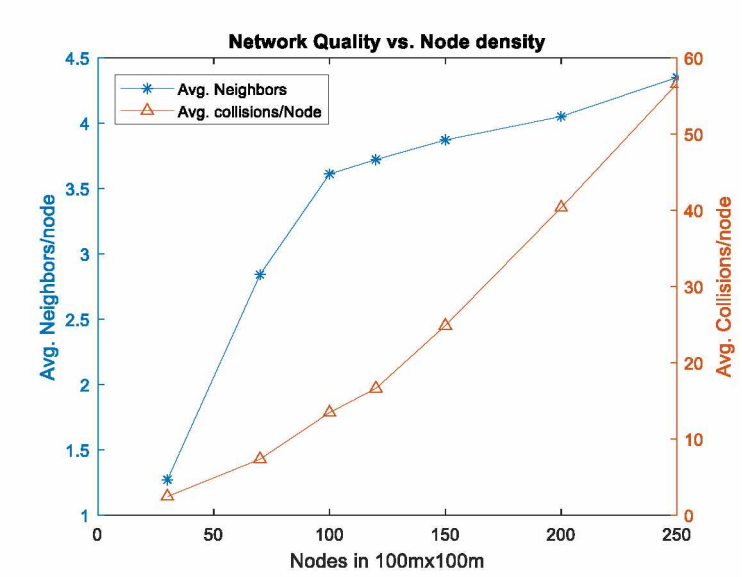



*Figure 9: Connectivity and collisions as function of network density*

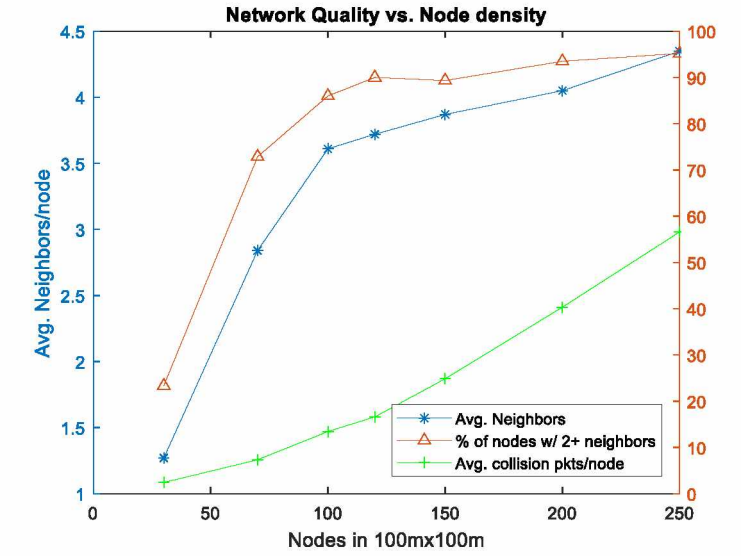



*Figure 10: Neighbor density and collisions vs network density*

From those observations, we selected 150 nodes in a 100 × 100 m area as the ideal density because it offers a consistent and high connectivity while having relatively low collisions. Over 150 nodes, the number of collisions grows significantly without necessarily increasing connectivity. Below 150 nodes, on the other hand, connectivity is too low. Even though all nodes have a connection, they do not have enough neighbors to allow for multiple paths, especially towards the edges.

Figure 11 shows an example of a random deployment of 150 nodes in a 100 × 100 m field, generated in Castalia. In Table 1, we collected the average number of neighbors per node, collisions per packet, and retransmissions for three randomly seeded deployment of 150 nodes. This experiment was intended to show that the network parameters remain fairly consistent and that they are independent of a specific deployment.

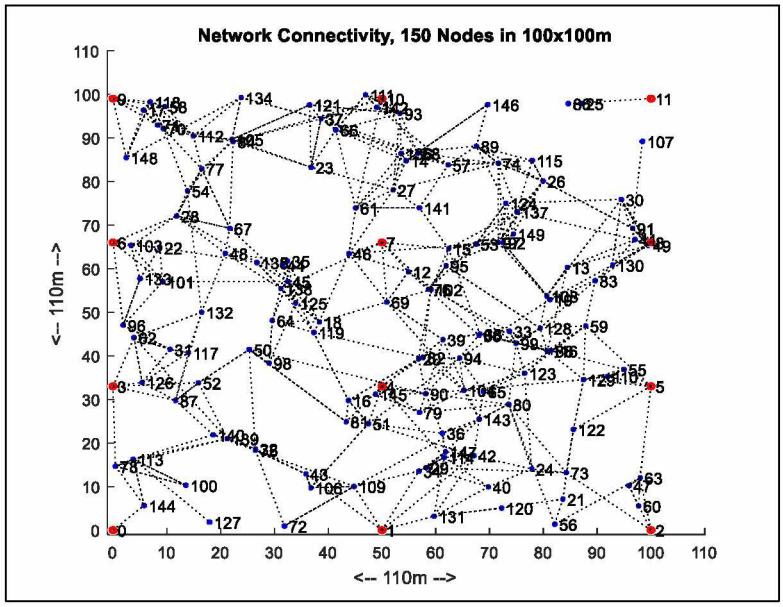



*Figure 11: 150 nodes randomly deployed in 100×100 m field*

*Table 1: Network connectivity performance for 3 random deployments of 150 nodes in 100×100m*

| | Avg. Neighbors per node | Collision packets | Retransmissions per packet |
|---|---|---|---|
| seed 0 | 3.87 | 24.36 | 2.64 |
| seed 1 | 4.06 | 23.57 | 2.39 |
| seed 2 | 4.09 | 21.23 | 2.88 |

It is important to point out that an improved algorithm to find neighbors can yield even better connectivity for densities higher than 120 nodes per 100 × 100 m area. Currently, the neighbor detection algorithm that was used is relatively simple because finding neighbors was not the main focus of the study.

Although we evaluate our network density in terms of communication coverage, it also has to meet or exceed the requirement of sensing coverage. But as we discuss in the following section, the required

density for sensing coverage in agricultural applications is usually much lower than that of communication coverage. In other words, if the density is good enough for communication coverage then it should be more than enough for sensing coverage.

### 4.4 Network lifetime vs active sectors

Before we explain how routing works, let us define network lifetime in our context. Keeping in mind that the goal of our sensor network is to monitor an agricultural field, we need the entire area to be covered by active sensors, i.e., be able to collect data and to communicate with at least one data hub (sink). The network is purposefully highly dense to allow coverage redundancy so that it can afford to lose several nodes and remain active as long as those nodes are not all in one area. So we divided the network into sectors and assess its lifetime based on how many of those sectors are active.

A sector is considered active when it has active nodes above a certain threshold. This threshold is determined by the user according to the desired coverage. In our network, we consider a sector to be alive if it has at least 40 active nodes. 40 nodes, if placed in the best possible arrangement in a sector (66.67 × 66.67 m) such as in Figure 12, would theoretically give us a coverage of 10.5 m per sensor node which is very close to our desired coverage of 10 m. In typical agriculture applications, 10 m is a high resolution; in the study done in [47], sensors were deployed 25 m apart. This paper also quotes another study according to which 1 – 16 soil samples per hectare or 30 – 100 m sampling distances are considered acceptable for PA. Even if we used these numbers for our coverage, our network would be disconnected due to the short range of the nodes' radio.

A higher threshold than 40 can be used if the user desires more precision, however, the user needs to keep in mind that since nodes are randomly deployed, some sectors start off with fewer nodes than others and that might reduce the lifetime of those unlucky sectors.

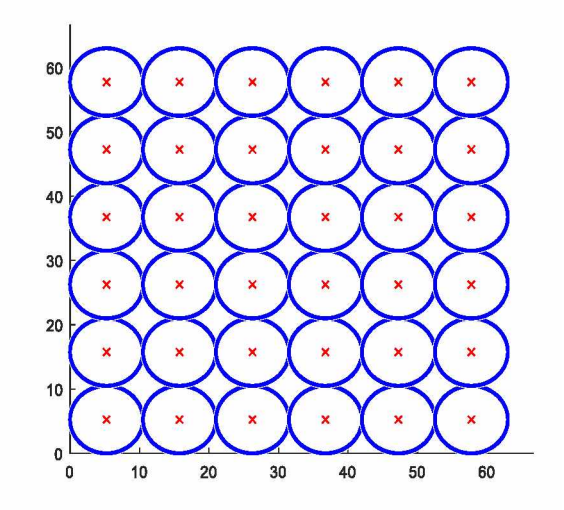



*Figure 12: Best case scenario for deployment in a sector*

Hence, the goal of our routing scheme is to keep as many sectors alive as long as possible, not necessarily to keep every node from dying. Besides, we value connectivity more than activity because a disconnected node, unable to reach the hub, is not useful even if it has a full battery.

### 4.5 Lifetime-maximizing routing

In order to maximize the lifetime of our network, we built a load balancing routing. Similarly to the algorithms in [10] and [11], our algorithm uses multiple routes to move data packets across the network.

In multi-hop communication, one of the fundamental techniques to save energy is for nodes to find the shortest route to the hub and continuously use it for data communication. This type of routing is called static routing. One example of static routing is shown in Figure 13, where node D found the route along nodes C and B to be the shortest. As we can see, there are other nodes using the same route. What eventually happens is that nodes on the shortest route (C and B in this case), use up their batteries and die. As a result, all the nodes that were using that route are left disconnected from the rest of the network. Static routing leaves the network with a wide disparity in the energy levels of the nodes [32].

There is another way of routing packets to the hub. Instead of using one short route, nodes can find multiple routes to the hub and dynamically choose the one to use when there is a new packet to send. In an example of dynamic routing shown in Figure 14, node D has three routes that it can use. When each nodes finds all the possible routes, the result is a web network of interconnected nodes. The advantage of this type of routing is that it preserves the network connectivity much longer than static routing. Since multiple routes are used, nodes on the shortest route will live longer. Besides, in case

nodes on one route die, nodes are not automatically disconnected because they might have other alternative routes.

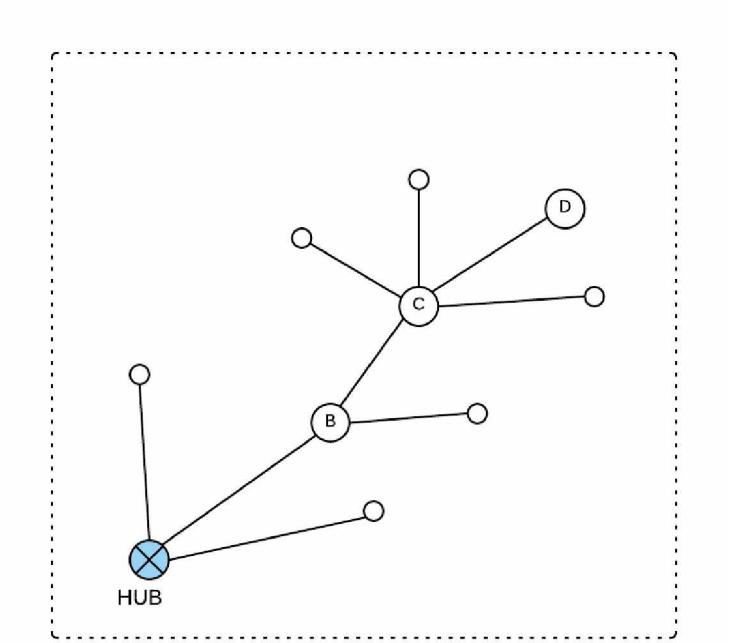



*Figure 13: Static routing*

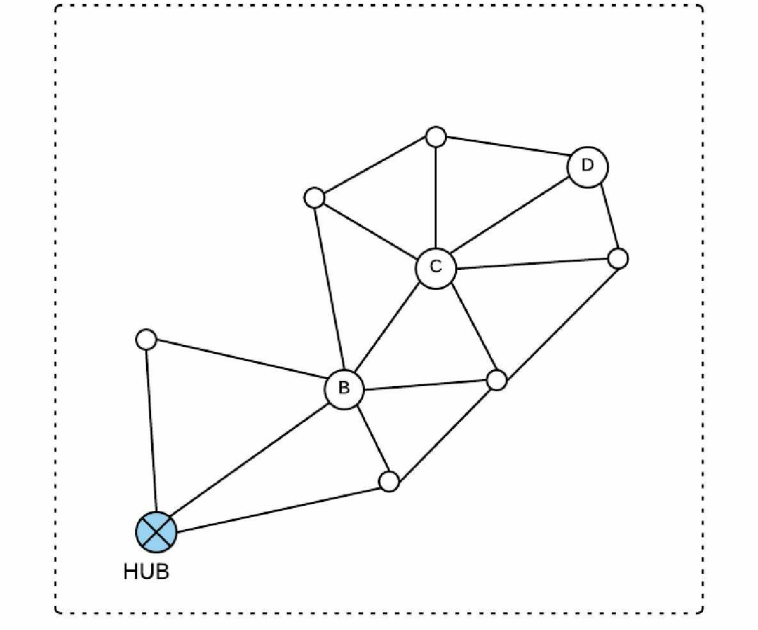



*Figure 14: Dynamic routing*

Route selection is based on the communication cost, which plays a crucial role in load-balancing routing, as we will see in the results section. The cost determines the potential of a node to carry packets. It is important to keep updated cost information across the network and to do that nodes regularly compute their own cost and share it with their neighbors.

The following are the major steps in the functionality of our algorithm:

- Each node computes its communication cost and updates it every 10 s.
- Each node finds as many neighbors as possible and builds a routing table. Each entry in the table has 4 fields – node ID, total packets, successful packets, and the cost. This step is very important because the algorithm performs better when given more routing options.
- To send a packet, nodes select the route with the lowest cost in the table. In return, they receive an acknowledgement containing the updated cost for this specific route.
- Each node regularly checks the status of its routes and removes from the table those with a poor link quality (<30%). The link quality of a route is calculated based on the total and successful packets that it carried.
- When the route table is empty, the node sends beacons to find more routes.

The cost function is a very crucial part of any load balancing algorithm and it is where algorithms differ. The main objective of the cost function is to indicate which nodes are in critical condition (high cost) and need to be spared or conversely, which nodes are in good condition (low cost) and open for communication. To prove the effects of the cost function, we experiment with four cost function types:

1. Static $Cost = C$ (constant cost or simply the hop count)
2. Exponential form: $Cost = A * e^{(B\frac{E_i}{R_i})}$
3. Power form: $Cost = (\frac{E_i}{R_i})^p$
4. Simple Ratio form: $Cost = \frac{E_i}{R_i}$

Where $Ei$ and $Ri$ are the initial and remaining energy respectively for node i. Parameters $A$, $B$, and $p$ are user inputs; in our experiments, we decided on the values A = 1000, B = 50, and p = 10. These values were experimentally proved to provide optimal results through multiple simulations. For the power form, we started with a high value of 50 as suggested in [48] but we discovered that, with a power of 50, the cost value quickly grows beyond the capacity of a 32-bit word (long integer) used in most low-power 16-bit microcontrollers. We then settled for a lower power, 10, which according to our results, performs nearly as well as 50.

### 4.5.1 Distance from Nearest Hub

In this stage, secondary nodes are trying to determine how far they are from the nearest hub (primary node). In this network distance refers to the number of hops it takes to reach a destination. So if node A is three hops away from node D, it means that there are two intermediate nodes between node A in question and node D, as shown in Figure 15 below.

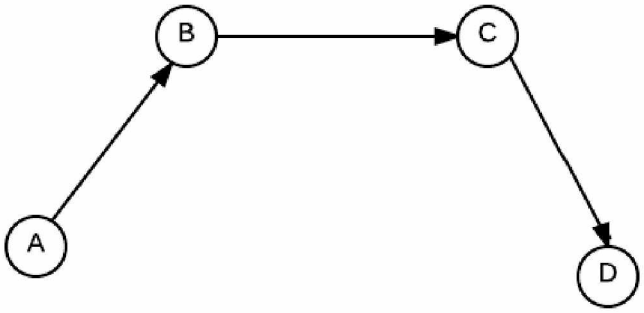



*Figure 15: Distance measured in hops*

In order to determine the distance, each of the secondary nodes broadcasts a few packets looking for a hub. The hubs, upon receiving that message, respond with an acknowledgement. Every node that receives an acknowledgement from a hub sets its distance to 1, meaning it is 1 hop away from the hub. Nodes that are not able to hear from the hubs start looking for neighbors who found a hub. If a node finds a neighbor that is connected to a hub, its distance is set to 2. This process continues until each node has a path to the hub and knows its distance from the hub.

Many of these steps of looking for a hub or a connected neighbor are repeated 2 to 4 times to increase the likelihood of finding the hub or the neighbor. In addition to that, every node or hub that receives a request from a neighbor waits a random time, between 0 and 5 s, before it replies to avoid collision. The idea here is that if an unconnected node broadcasts a packet and one hub and one connected node both hear and immediately respond, the two responses will most likely collide and the unconnected node might not hear from either. But if the hub and the connected node each wait a short random time and they respond at different times, the recipient node will most likely receive the two packets and decide which one to connect to; in this case the node should select the hub. This is an important feature in the application that is used to avoid collision in any communication that involves more than two nodes.

In this step, each node also saves the ID of the node or hub it is connected to and that serves as the first hop in the shortest path to the hub. This shortest path is the one used for transmission in static routing.

### 4.5.2 Building Routing Table

After determining their relative location, nodes build a table of routes that they can use to send data to the hub. This table plays a vital role in the routing process as we will see in the routing section. Each route entry in the table has the following attributes:

- A Route ID
- Total packets [sent through this route]
- Successful packets [sent through this route]
- length
- Cost [to Send through this route]

A route was implemented in C++ as a struct data type and the routing table is vector of routes, as seen in Figure 16.

```
struct route {
    int routeID;
    int tot_pkt;        //total packets sent on link
    int sx_pkt;     // Successful packets on link
    int length;     // Route length in hops
    double cost;

    bool operator==(const route& r) const;
    bool operator<(const route& s) const;
};

std::vector<route> myRoutes;
```

*Figure 16: C++ implementation of a route and the routing table*

In order to build the route table, nodes send out 4 FIND_NEIGHBORS beacons every 5 s. Each node that receives such a packet responds with a FIND_NEIGHBORS_ACK which indicates that this neighbor is within reception range and can be used for routing. The initiator then proceeds as follows:

- Create and add a new route using the information in packet: route ID is set to packet source ID, total packets and successful packets are set to 1, length to packet version, and cost to packet data.
- Update the route with new information if it already exists in the table.

- Forward the *COST_INFO_PKT* to neighbors with data including node's distance to hub and communication cost.

Nodes that have less than 2 routes, send 3 additional FIND_NEIGHBORS packets to find more routes. The number of routes is a major factor in the performance of load sharing routing; the more routes available the easier to spread the load and preserve battery.

### 4.5.3 Data Routing

Nodes generate a data sample every *sample_T* sec, a sampling period set by the user. When a node has a new data sample, the following steps are taken to deliver it to the hub:

**Step 1:** The node sorts the route table mentioned in section 4.5.2. As mentioned before, the routing table was implemented as a C++ vector which has a *sort()* member function.

In order for the *sort()* function to work, elements in the vector need to be comparable using the less than ('<') sign. Elements in the routing table are of type route, defined in Figure 17, and using the overloading feature of C++ classes, the '<' operator was implemented, Figure 17, to allow routes to be comparable and therefore sortable. Another advantage of implementing the comparator is that we get to choose the factor of comparison. In this case we compare routes based on their cost, hence the first entry in the sorted table is the route with the lowest cost.

```
bool route::operator==(const route& r) const
{
    return this->routeID == r.routeID;
}
bool route::operator<(const route& s) const
{
    return this->cost < s.cost;
}
```

*Figure 17: C++ Implementation of the '==' and '<' operators for class route*

Figure 17 also shows the implementation of another overloaded operator, '==', which is used to search the routing table for existing routes.

**Step 2:** The node sends a data packet through the first route in the table, i.e., the one with the lowest cost and increments the *total packets* field. Figure 18 shows the implementation.

**Step 3:** If no acknowledgement is received within a second, the node retransmits the packet and waits for another second.

**Step 4**: After two unsuccessful re-transmits, the node re-updates the route table to find a new route and then repeats Step 3.

**Step 5:** If no acknowledgement is received after five trials, the node stops sending the packet which is considered a failure (the packet is lost).

Every time a node sends a packet via any route it increments the *Total packets* field of that route. The packet is also added to the packet table containing all packets sent or forwarded.

**Step 6:** When an ACK is received, the node marks the status of the packet in the packet table as successful. It also increments the *successful packets* field of the route that was used. The node calls the *updateLinkQuality()* function, see Figure 18, with the route ID and a number 0 or 1. 0 means a new packet has been sent via this route, 1 means an acknowledgement was received on this route.

```
void TwoLayerNetwork::updateLinkQuality(int routeID, int x){

    std::vector<route>::iterator r_ptr;
    route r({routeID, 1, 1, 1, 0});
    r_ptr = find(myRoutes.begin(), myRoutes.end(), r);
    if(r_ptr == myRoutes.end())          // Route doesn't exist in my
    route table
        return;
    if(x == 0)
        r_ptr->tot_pkt++;
    else if(x==1)
        r_ptr->sx_pkt++;
}
```

*Figure 18: C++ implementation of link quality update (step 6)*

When a node receives a packet from a neighbor it sends back an ACK immediately and then follows steps 1 – 6 to forward the data packet. Forwarding continues until the packet reaches a hub. The ACK packet in this protocol is not only used for reception confirmation but also to update the route table. ACK packets contain the most up-to-date cost information which the ACK receiver adds to the route table as the new cost of this specific route. Besides, when a node receives an ACK, it increments the *successful packets* field of the route where the acknowledged packet was sent.

The goal of using a route table is to allow each node to have multiple options when routing a data packet. As the authors of [32] suggest, protocols that only use one optimal route burn the energy of the nodes on that route faster and that leads to a large energy disparity in the network. The route table also allows a node to choose the optimal route, i.e., one that preserves the longevity of the network. More

specifically, a node chooses the route with the lowest cost. As we will see in the following section, the cost function was designed so as to preserve energy and to avoid bad links.

### 4.6 Cost Function

One of the characteristics of routes in Section 4.5.2 is communication cost or simply cost. Cost function is a very crucial part of any load balancing algorithm as we will show in the results section. The main objective of the cost value is to indicate which nodes are in critical condition (high cost) and need to be spared or conversely, which nodes are in good condition (low cost) and open for communication. Communication cost can be calculated as a function of various parameters, such as energy, link quality, and number of neighbors. For this study, we only considered energy status (remaining battery energy) and link quality.

#### 4.6.1 Residual Energy

If the goal of our routing protocol is to manage the energy resource carefully so as to extend the network's lifetime, we must consider energy in our choice of a route. Hence battery status or residual energy is at the core of the cost function. Energy is also used in other similar studies such as [10] [11] that use a cost metric function to find the best route. This cost is shared among neighbors to help them determine the cheapest route to the hub. Although many studies used the energy in the cost function to create load-balanced networks, the actual form of the cost function varies tremendously [11]. One study for example, suggested an exponent function [48] as seen in Equation 3,

$$c_{ij} = T_{ij}^{a} R_i^{-b} E_i^{c}, \tag{3}$$

where $T_{ij}^{a}$ is the transmission cost between nodes i and j, $R_i$ and $E_i^{c}$ are the residual energy and initial energy respectively for node i.

The form of the cost function is very important because it affects the performance of the routing. To show that, we comparatively study four types of cost functions: static, ratio, power, and exponential

**a. Static routing**

In static routing, nodes select the shortest route (Section 4.5.1) and only use that route to communicate with the hub. As opposed to dynamic routing, routes to the hub do not change and there is no cost function hence the name 'static routing'.

The following three routing types are dynamic since nodes use different routes to send packets.

**b. Ratio form**

The ratio form is the simplest of the cost function we used but effective enough to show the power of load-sharing routing. The actual function is shown below:

$$Cost = \frac{E_i}{R_i}, \tag{4}$$

where $Ei$ and $Ri$ are the initial and remaining energy respectively for node i. Those are used in the power and exponential cost functions as well.

**c. Power or exponent form**

The power form was inspired by the work of Chang and Tassiulas who determined that a using a high exponents for $Ei$ and $Ri$ in Equation 5 gives the best performance [48]. The actual function resembles the ratio form but with a power:

$$Cost = (\frac{E_i}{R_i})^p. \tag{5}$$

As suggested in [48], we started with a power of 50. But we quickly noticed a problem; the ratio $(\frac{E_i}{R_i})^p$ rises to a very high number, too big for a 32-bit word used in most microcontrollers. For example, when the residual energy reaches 50%, the cost in Equation 5 would be $2^{50}$ while the largest number that can fit in a 32-bit word is $2^{32}$-1. In other words, using a high exponent is not practical in actual implementations, even though it performs well in simulation. Eventually, we settled on the power of 10 because it does not raise the cost quite as high and yet it performs quite as well as the power of 50 in terms of network lifetime.

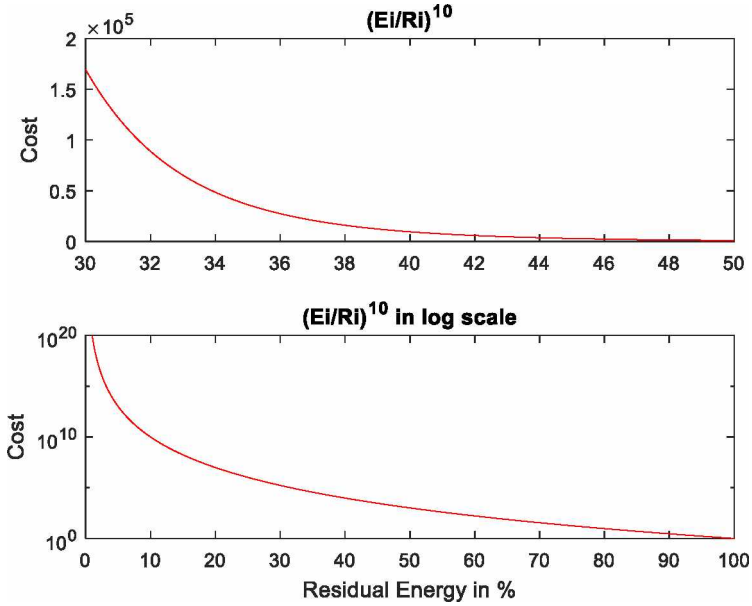



*Figure 19: Power cost function characteristics*

The power function increases the performance of the load-balancing routing by amplifying the battery effects on the cost. As we see in Figure 19, the power function rises very fast for a slight change in battery levels. When the remaining battery drops from 50% to 30%, the cost rises by a factor of $10^5$, as seen in the top half of Figure 19. This property of the power function is more accentuated at lower levels of the battery as we see in the log-scale version of the cost function, Figure 19 (bottom-plot).

### d. Exponential form

In light of the possible shortcomings of the power cost function described above, we wanted to look at another form of a cost function that allows the cost to gradually increase as the energy decays. Exponential and sine functions allow small changes in the input to make a significant effect on the output end [11]. Using a similar logic, we devised a cost function that allows small changes in the energy to affect the choice of a route.

$$Cost = A * e^{(B\frac{E_i}{R_i})}. \quad (6)$$

This cost function provides more control on the growth rate, as we can see in Figure 20, which shows plots of Equation 6 for different values of parameters $A$ and $B$.

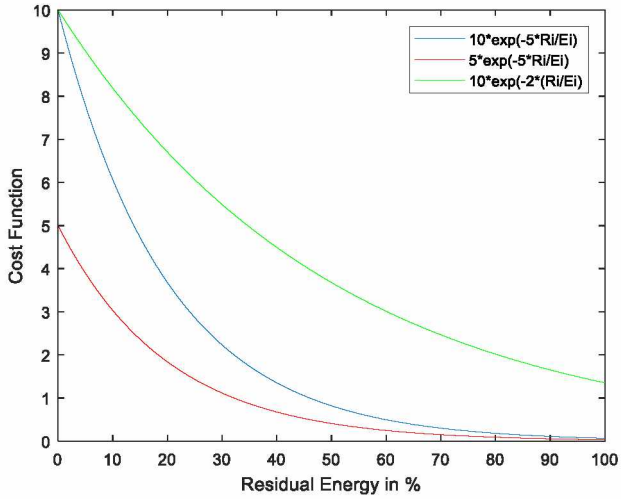



*Figure 20: Cost function in exponential form*

This new form of cost function that uses the exponential function has a more gradual trend than the power form. The cost starts to increase slowly as the energy drops and once the energy gets too low (<30%) the cost grows very fast. In addition to that, the two parameters *A* and $B$ make the exponential cost function more flexible. Parameter *A* determines the peak or the maximum value of the function while $B$ determines how fast the function grows. For example, the red function in Figure 20 uses a low value of *A* (5) compared to the blue one; hence, it peaks at a low value. The green function on the other hand, has a lower value of $B$ (2) which is why it starts to rise earlier but slowly compared to the blue one.

Besides its flexibility, the exponential cost function outperforms the power cost function in terms of packet delivery rate, as we will show in the next chapter, section 5.6. And we believe that by carefully tuning the parameters in equation (6), the exponential can perform as good, or even better than the cost function in energy efficiency.

### 4.6.2 Link Quality

Link quality (LQ) is often left out in load-balancing routing protocols; that is the case for studies in [10], [11], and [33]. This can probably be justified by the fact that in those studies, the authors assume the link quality to be generally high for the entire network in which case the LQ would not make much difference. However, that is not the case in farming applications; LQ can vary tremendously from one link to another due to obstacles such as trees, landscape, or farming machinery. Therefore, it is important to consider it in the route choice process and so, in this section, we study how using the LQ in the cost function affects the network lifetime.

As mentioned in section III, nodes keep information about their neighbors and part of that information is the total packets and successful packets. These two fields are used to compute the link quality of each route as follows:

$$Link\ Quality\ (LQ) = \frac{successful\ packets}{total\ packets}\ . \tag{7}$$

We incorporated the link quality into the cost function to allow nodes to choose not only the neighbor in good energy conditions but also one with the best link. The new cost function with link quality (LQ) is shown below:

$$C_{LQ} = C_i(\frac{1}{LQ})^{pow} + C_j\ . \tag{8}$$

where $C_i$ is the energy-based cost of the transmitting node, $C_j$ the cost of lowest-cost neighbor, $LQ$ the link quality of the link i-j, and $pow$ the variable weight for LQ. Also note that the cost is inversely proportional to LQ, which means that a poor link quality leads to a high cost.

### 4.7 Maintenance

#### 4.7.1 Route Table Maintenance

Every 10 s, the node (or, in Castalia terminology, application) runs two maintenance functions: one for the route table and another one for the battery. The route table maintenance consists of removing routes with poor link quality. In the current application, any route with a link quality below 20% is removed from the table. In case the table becomes empty, a flag is set and the node starts to look for new routes. A node cannot send data packets if the route table is empty. Figure 21 shows how this feature was implemented in Castalia.

Poor routes, i.e., those with a link quality below 20% are not only removed but they are added to a bad route table so that in the future when the node is looking for new routes it would not add one of those bad routes.

```
void TwoLayerNetwork::routeTableMaintenance(){

    std::vector<route>::iterator rt;
    if(myRoutes.size() > 0)
    {
        for (rt = myRoutes.begin(); rt < myRoutes.end(); rt++)
        {
            if((double)rt->sx_pkt/rt->tot_pkt < 0.2)        // 1
            {
                myRoutes.erase(rt);
                badRoutes.push_back(*rt);
                collectOutput("Remove route", rt->routeID);
            }
        }
    }
    else if(myRoutes.size() == 0 && connected < 2)
    {

        connected = 1;          // No routes available
        findNeighborAtmpts = 0;
        setTimer(FIND_ROUTE, 2);
        cancelTimer(TX_DATA_PKT);

    }
}
```

*Figure 21: Route table maintenance C++ implementaion*

Maintaining the route table allows nodes to choose from a list of good routes and therefore avoid bad links that drain energy through retransmissions. For example, a neighbor node may have a lot of remaining energy and therefore look like a good route while the link to it is very poor. This kind of node should be avoided unless it is the only possible route. Maintenance also helps keeping the table size small and easily manageable.

### 4.7.2 Battery Maintenance

In this stage, the application updates its cost using one of the cost functions in section 4.6.1. Prior to computing the cost, the route table is sorted to make sure the cheapest route is selected. The costs of routes keep changing due to the changes in energy levels of the corresponding nodes as well as their link quality. It is important for a node to maintain the most up-to-date cost so that the neighbors' route tables are based on accurate information.

## 4.8 Energy Model

### 4.8.1 General Concept

The cost function in section 4.6 relies mainly on the residual energy of the node, and it is therefore crucial to have an accurate energy model. Moreover, the goal of this study being to design a lifetime-maximizing routing protocol, an accurate energy model is paramount to obtain accurate results. Similar

network lifetime-maximizing studies such as [10], [11], and [33] use simplistic energy models where energy is only consumed during the TX and RX modes of the radio. That assumption might work for highly active networks where sleep time is relatively short. But that is not the case for our network; nodes spend a lot of time sleeping and only wake up for a few microseconds to send or receive data. Let us look at a concrete example to illustrate this point:

Consider the CC2420 [46] radio that is used in Castalia:

*Table 2: CC2420 Radio main parameters*

| CC2420 Radio | | |
|---|---|---|
| **Features** | **Value** | **Unit** |
| **Transmit Rate** | 250 | kbps |
| **TX current** | 8.5 at -25 dBm | mA |
| | 9.9 at -15 dBm | mA |
| | 11 at -10 dBm | mA |
| | 14 at -5 dBm | mA |
| | 17.4 at 0 dBm | mA |
| **RX current** | 18.8 | mA |
| **Sleep current** | 20 | uA |

Consider an application that transmits 5 packets and receives 4 packets in 50 s and each packet is 50 bytes long. Using the parameters in Table 2, we can determine that the node will spend 8 ms in TX mode, 6.4 ms in RX mode, and the rest in sleep mode. The total energy consumed is 1.2 mAs, 82.7% of which was consumed in sleep mode. This shows why a simplistic energy model that only considers energy spent in RX and TX is not reliable. The importance of sleep mode becomes even more significant for networks that live for a long time, weeks or months. Our energy model considers both the energy used to send and receive packets as well as the energy dissipated in sleep mode.

Besides data transmission and reception, the energy of a node is dissipated by the wake-up receiver (see section 4.6), the radio in sleep mode [46], and the microcontroller both in active and sleep mode. For the microcontroller, we used TI MSP430F5438A [49]. Taking into account all the mentioned factors, the following equation can be used to determine the energy consumed in one frame (50 s):

$$E = \left(n_{tx}.L.\frac{8}{bitRate}\right).I_{TX} + \left(n_{rx}.L.\frac{8}{bitRate}\right).I_{RX} + T_{sleep} * (I_{r_{sleep}} + I_{WuR} + I_{MCU_{sleep}}), \quad (9)$$

where $n_{tx}$ and $n_{rx}$ are the number of packets sent and received respectively, $L$ the packet size in bytes, $bitRate$ the radio's bit-rate in bits per second, $I_{TX}$ and $I_{RX}$ the TX and RX current, $I_{r_{sleep}}$ radio current in sleep mode, $I_{WuR}$ wake up radio current, $I_{MCU_{sleep}}$ microcontroller sleep current, $T_{sleep}$ sleep time.

### 4.8.2 Energy model testing

Now, let us test the accuracy of our energy model. To test the energy model accuracy, we simulated the network for 10,000 s with each node generating a new packet every 50 s. With those settings, the following results were obtained:

- Average Consumed Energy (per node), $E_{avg}$ = 327.79 mAs
- Average TX packets (per node) $TX_{pkt}$ = 304.41 packets
- Average RX packets (per node) $RX_{pkt}$ = 640.82

With these numbers and the knowledge of the current draw in different modes, we can verify that the consumed energy is equivalent to the activity of the node:

1. From $E_{avg}$ we can derive the average current draw: $I_{avg} = E_{avg}/10000s$ = *32.8 µA*
2. In sleep mode, the node draws $I_{sleep}$ = 28 µA (see section 3.5.1)

Consider one frame to be 50 s long since each node sends a packet every 50 s.

3. The average number of packets sent in one frame is: $TX_{pkt}$ × 50 / 10000 = 1.52,
4. So we can calculate the average TX current draw: 11mA × (1.52 × 105 bytes × 8 bit/byte / 250 kbps) / 50 s = 1.1 µA
5. The average number of packets received in one frame is: $RX_{pkt}$ × 50 /10000 = 3.2
6. Then the average RX current draw in a frame is:
   18.8mA × (3.2 × 105 bytes × 8 bit/byte / 250 kbps) / 50 s = 4 µA
7. The total current draw can then be calculated as (28 + 1.1 + 3.2) µA = *33.1 µA*

The two current values, one from the simulation results (1) and the predicted one (7), are very close to each other, only 0.3 µA apart which confirms the accuracy of our energy model.

Furthermore, we wanted to see how the energy is consumed over time, so we ran the simulation with various time limits and the results are shown in Figure 22.

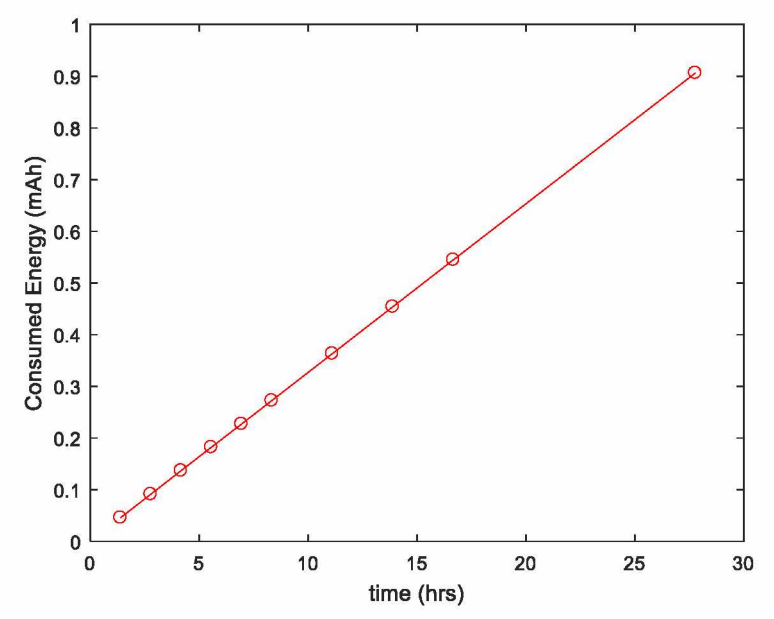



*Figure 22: Energy Consumption over time*

The first observation is that the energy is drawn linearly over time. That might seem suspicious at first to some readers, but let us once again consider the network duration to be divided into 50 s frames and that on average the same number of packets are sent and received in each frame. That means that the same amount of energy is drawn during each frame and therefore increasing the network duration by a factor of the frame size will increase the consumed energy by the same factor.

The linear trend of the consumed energy also allows us to run the simulation for only a relatively short amount of time and obtain credible results since the behavior does not change over time significantly. Running simulation for a short time has two advantages:

1. Short simulations run fast and are less likely to crash. The longest simulation that was run successfully was 200,000 s which is about two and half days. Longer simulations not only take too long but they often crash.
2. Short simulations with low initial energy allow us to observe the end behavior of the network as nodes start to die. This is very important because it reveals the efficiency of the routing protocol, how well the network was preserved which is the main focus of this research. More details are provided in chapter 5 - results and discussion.

Let us clarify that although our energy model was deemed good enough for evaluating our routing protocol, it does not model the correct behavior of real batteries in wireless sensor nodes. The

measured average current of 32.8 µA suggests that a typical AA Alkaline battery with 1800 mAh capacity [50] would last 54878 hours or 6.26 years. These numbers are far from the actual battery lifetime and that is because many factors that affect battery life were not taken into account in the modeling. Here are some of those factors:

- The MCU LPM3 current used in the model [49] does not include the DC/DC converter which consumes a good portion of the battery power. Park et al. show that the DC/DC converter, found in almost all microcontrollers, consumes about 30% of the power it draws from the battery [51]. Under certain conditions, only about 70% of that power is delivered to the microcontroller and its peripherals.
- The battery energy capacity (mAh) assumes a constant current draw, but in reality, sensor nodes draw a variable amount of current depending on the tasks being executed. Various studies have shown that batteries behave differently depending on how they are used. Feeney et al. show that the battery capacity is affected by the current rate and the duty cycle of the load activity [52]. For example, they show that a circuit load drawing 4 mA drains the battery four times faster than 1 mA but it also reduces the capacity by 20% [52]. According to the study presented in [51], drawing power at a duty cycle of 1:5 increases the expected battery capacity by 250%. The 1:5 duty cycle was accomplished by running the node in TDMA, where the node draws more current in receive mode and much less current in sleep mode, to mimic typical node activity in WSN.
- The microcontroller current is not considered in the energy model because it is fairly complicated to determine how much time the MCU spends in active mode, without knowing the details of the node's functionality. However, it is important to note that running complex functions such as our cost function, a 16-bit MCU might take much longer time than the radio spends transmitting or receiving data. With that, the power consumption from the MCU can be comparable if not more than the radio's despite its lower current.
- Our energy model does not consider energy consumed by the radio during wake-up time. Moreover, the microcontroller wakes up before and more often than the radio to execute other tasks such as reading sensors, data processing, and data packaging.

### 4.8.3 Current profile

Current profiles, such as the one in Figure 23 below, are useful to understand the current draw (or power consumption) of a sensor node during the various modes of activity. Figure 23 shows the current profile of an active node in our network during a 20 s period. The node can go in 4 different modes: sleep mode, active CPU mode, transmit (TX) mode, and receive (RX) mode. RX and TX are the most power hungry with 17.4 mA and 11 mA current draw respectively. During the 20 s period, each node receives on average 2.45[1] data packets and 2.31 acknowledgement (ACK) packets. The node also sends 2.45 ACK packets and 2.7 data packets including one of its own and the rest from neighbors. Sending or receiving a data packet takes 3.2 ms while an ACK packet takes 352 µs. That means that the node is in TX or RX mode for a very short time which saves energy because those two modes consume the most power.

Before sending and after receiving a packet, the node wakes up from sleep mode to process the data. Unfortunately, due to the simulator's limitations, we were not able to determine how long the node stays on or the power drawn to process the data. It is also important to mention that the hand-drawn current profile in Figure 23 is not to scale; for example, the gap between packets during which the node is in sleep mode is much longer than the time it takes to send or receive packets. Also, the sleep mode current (28 µA) is very low compared to other modes of activity such as RX and TX. Besides, the instance where the node sends a packet, such as $t_1$ and $t_2$, are random which does not affect the amount of power consumed. Randomizing packet transmission helps reduce interference between neighboring nodes. 20 s after the node sends its first packet, it sends another packet and cycle repeats itself until the node runs out of power.

[1] To obtain the average number of packets and ACKs sent and received, we ran a 3000 s simulation and collected the total number of packets and ACKs transmitted then we scaled that number down to a 20 s period.

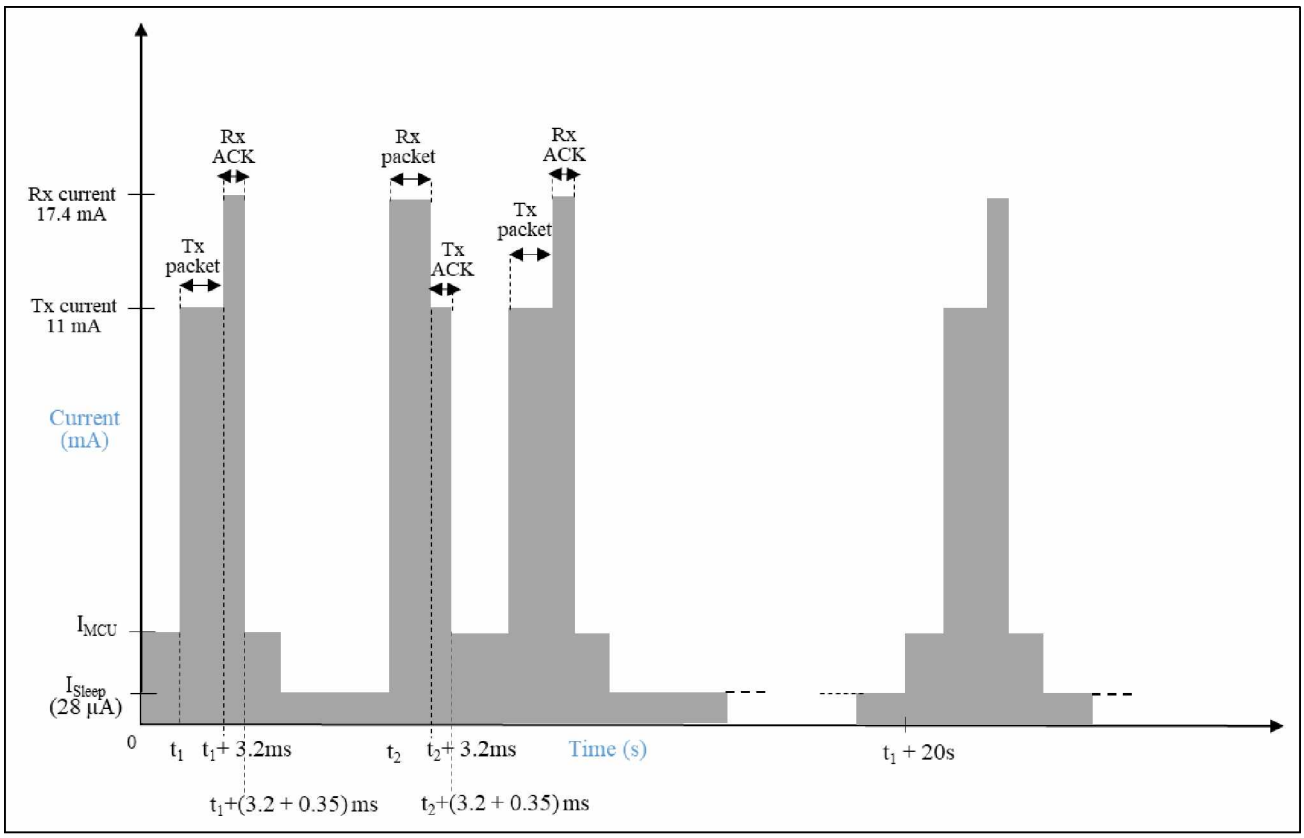



*Figure 23: Current profile of node in different activity modes*

In our experiment, each node sends a new packet every 20 s; this is a very short period compared to typical agriculture applications. For example, in the experimental study that introduced a sensor network in a horticulture farm in the semiarid region of Murcia, Southern Spain, the authors sampled and reported the temperature at an hourly rate [53]. We choose a shorter sampling period to condense our experiments within a short time frame because the simulation tool that we used does not allow long simulations; simulations longer than 10,000 s (or roughly 2.8 hours) did not run successfully. A shorter sampling period also increases the radio activity which in turn amplifies the effects of load-balancing routing. The large size of data packets, 100 bytes, was another attempt to increase radio activity. Increased radio activity is good for this study because it enhances the importance of load-balancing routing. It is the reason why we see a significant improvement in network lifetime for dynamic routing over static routing as we will show in chapter 5. Experimental results showed that low radio activity, due to small packets or large sampling period, reduces the effects of load-balancing routing on lifetime.

### 4.8.4 Wake-up Radio vs. Synchronous sleep schedule MAC

The radio draws the most power in a wireless sensor node, so it is important to keep the radio in sleep mode for as long as possible. There are two ways to do that: using a duty-cycling MAC protocol or using a low power wake-up receiver [54]. Each method offers certain advantages and poses its own challenges.

A wake-up receiver (WuR), such as the one on the Piconode [10], is a small low-speed radio that consumes very little power. WuRs are used together with more powerful and energy hungry radios to achieve asynchronous sleep schedule [55]. The WuR remains in idle listening mode while the main radio sleeps; when a wake-up call is received, the WuR awakes the main radio to handle data communication. Although they are always listening, WURs consume very little power; the AS3933 from Austria Microsystems (AMS), for example, consumes as little as 2.3 µA in idle listening [56]. It is important to mention however that the low power consumption of WuRs comes at a cost, they have low sensitivity and reduced range [54]. That is the first challenge of WuR, due to their limited receive sensitivity, they are not be able to catch some low-energy signals that the main radio would have detected. Another problem that comes with WuR is false wakeups [54] which happen when the WuR detects a signal on the channel and triggers the node to wake up, only to find out that the signal was intended for a different node.

Synchronous sleep or duty-cycling MAC Protocols are another alternative to save power. Unlike WuR, they do not require any additional hardware. Duty-cycling MAC protocols save power by putting the radio to sleep, only waking shortly and regularly to send data or to receive data [54]. On the downside, these protocols suffer from increased latency [54] because nodes can only communicate when they are not sleeping. For our network, we considered using an adaptive duty-cycling protocol, *TMAC*[2], as an alternative to WuR. Although it would have reduced the hardware complexity, TMAC led to a very poor throughput where only about 50% of packets made it to the hubs. The authors of TMAC predicted such behavior in [57], which they call early sleeping problem. As the name suggests, that problem is caused by nodes going to sleep before they can receive packets from neighbors. TMAC authors also believe that this problem may occur in asymmetric communication [57] and our network can be considered asymmetric since the packet flow is always from nodes to hubs. The suggested solution to that problem, implementing a *future request to send* [57], is unfortunately not implemented in Castalia [58].

[2] Timeout Medium Access Control, TMAC is implemented in Castalia

For this network, we chose to use wake-up receivers for the following reasons:

1. TMAC performed very poorly in terms of data packet throughput.
2. Removing the MAC layer allows the other network layers to work properly and to be evaluated easily [32]. This research is focused primarily on routing protocols, stacking another layer will certainly affect the functionality and performance of our routing protocol. For example, our proposed routing protocol uses packet acknowledgements to exchange cost information between nodes but TMAC already has its own ACK implementation, which cannot be used for the same purpose. Using both would have required disabling or considerable modification to one of the two layers.
3. Using wake-up receivers is the most compatible solution for a network with unpredictable communication pattern such as ours. Duty cycling protocols, such as TMAC, in their original form would not be compatible with this network.

# Chapter 5 Results and Discussion

In this section, we use simulation results to show how the network performance is affected by the cost function. First, we comparatively study the four energy-based cost functions mentioned above – static, exponential, power, and ratio and how they affect the network lifetime. Secondly, we add the link quality factor to the cost function and see how it affects network performance. Simulations were generated using Castalia, a WSN simulator based on Omnet++ [41]. Despite its many useful features, Castalia has limited tools to display results thus a combination of Python scripts and Matlab was used to translate the raw data from Castalia into meaningful results and easily readable graphs.

## 5.1 Residual energy and network lifetime

First, we ran the network with each of the cost function and observed the lifetime. The results, plotted in Figure 24, show that the power form performed the best while the static form was the poorest as we anticipated. The exponential and ratio forms performed better than static but not as good as the power form. The first sector in the static form became inactive after 2,690 s while it took 5,386 s for the power form, about twice as much time; Table 3 shows more details. These results also confirm the claim in [48] that using a high-power $p$ for the energy factor in the cost function gives the best results.

*Table 3: Time when first and last sector become inactive for each cost function*

| | First Sector | Last Sector |
|---|---|---|
| Static | 2690 | 3861 |
| Exponential | 4938 | 5202 |
| Power | 5386 | 5475 |
| Ratio | 4259 | 4353 |

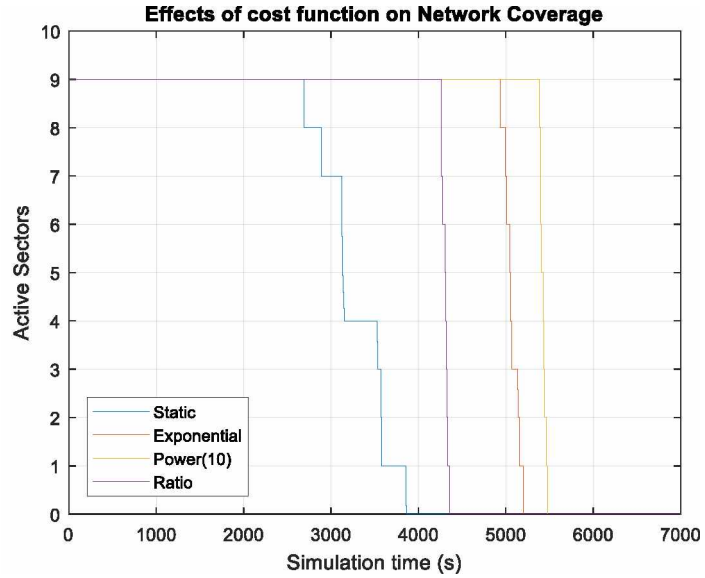



*Figure 24: Network coverage profile for different cost function*

In Figure 25, we see that the ratio, exponential, and power functions not only keep the sectors alive longer but also, they keep almost all nodes connected the longest as well. The active nodes' curve (red dashed lined, showing the number of nodes that are still active) follows the sector curve almost perfectly for the three dynamic (i.e., non-static) cost functions.

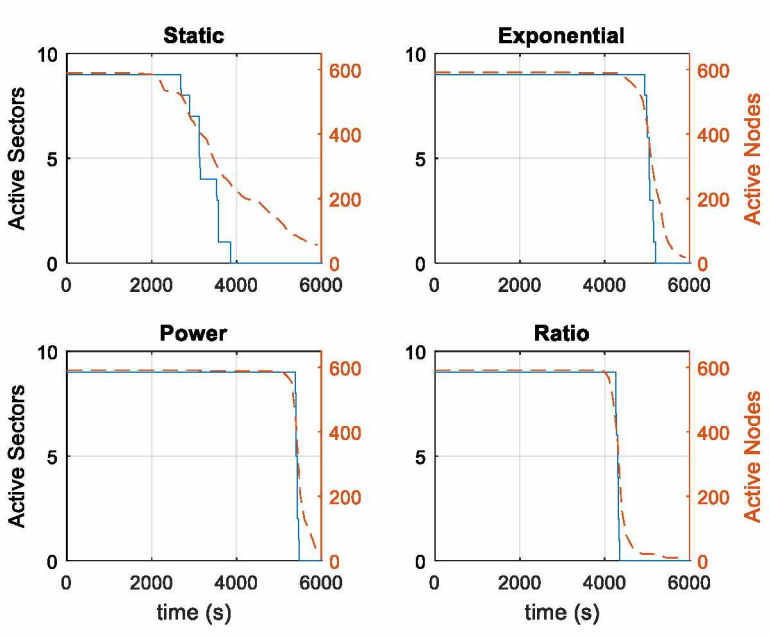



*Figure 25: Network profile (sectors and nodes) for each cost*

When we consider a node to be alive until its battery exhaustion instead of when it gets disconnected, the results are reversed. The static cost function preserves the majority of sectors and nodes for the longest time, as shown in Figure 26 and Figure 27. But nonetheless, the static form loses a few nodes earlier compared to other forms, (Figure 27). As we will prove later, in static routing, a few nodes carry the majority of the packet traffic and quickly exhaust their batteries leaving the network disconnected since routes are static.

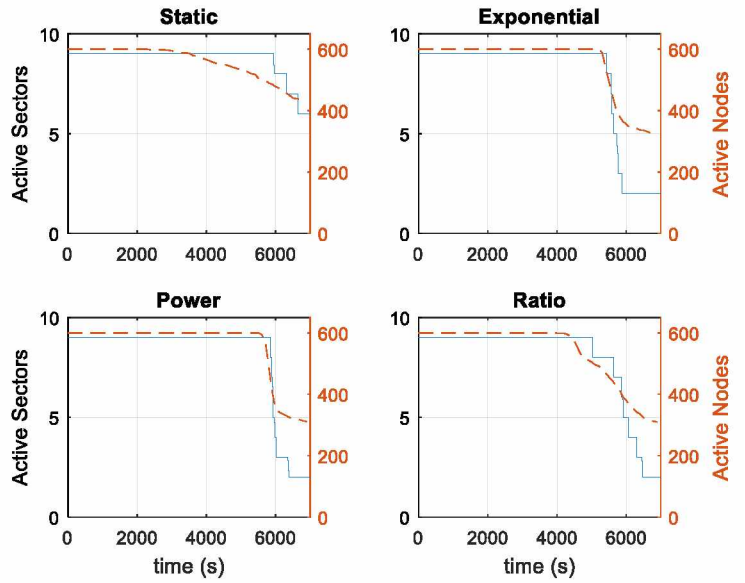



*Figure 26: Network profile (sectors) considering battery exhaustion*

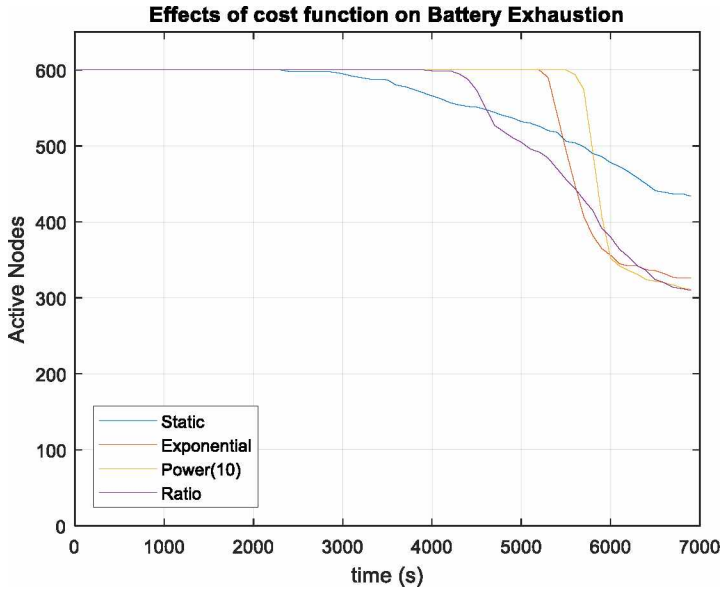



*Figure 27: Network profile (nodes) for battery exhaustion*

The following charts show the traffic load distribution among nodes for the four routing cost functions. The packet traffic distribution is important in understanding the difference in performance routing protocols because these protocols extend the network lifetime by evenly distributing packets.

In Figure 28, we see that the majority of nodes do not carry any traffic from their neighbors for the static cost. More than 300 nodes do not carry any traffic and another 100+ only carry traffic from one neighbor; those nodes represent more than 2/3 of the network. In other cost functions, such as the power form, the majority of nodes carry packets from 4 or 5 neighbors and only a few nodes (<50) carry traffic from one neighbor; those nodes are probably on the edge of the network. That is load balancing in action.

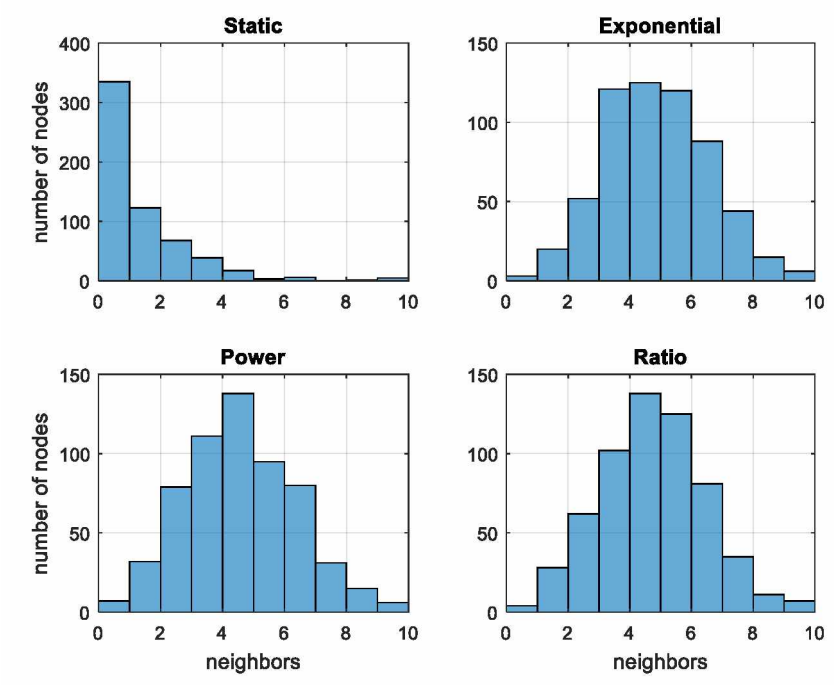



*Figure 28:The number of neighbors serviced by nodes for routing purposes*

Taking multiple routes means taking longer routes at times as we see in Table 4. Packets cover a longer distance on average to get to the hubs for the three non-static cost functions. That however does not necessarily result in poor network performance as we saw earlier.

*Table 4: Average distance coverage by packets*

| Cost type | Distance (hops) |
|---|---|
| Static | 2.46 |
| Exponential | 3.64 |
| Power | 3.35 |
| Ratio | 2.63 |

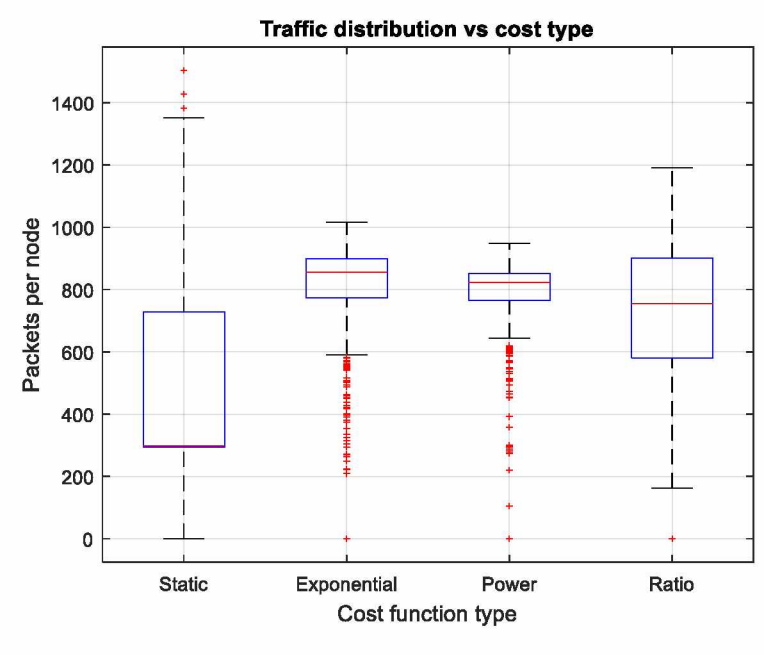



*Figure 29: Packet traffic distribution among nodes*

Figure 29 shows the distribution of transmitted packets among nodes. The size of the box and the length of whiskers indicates how closely or widely the packets are distributed among nodes. For example, static routing has the largest box and the longest whiskers indicating that packets are unevenly distributed among nodes. Half of nodes only send about 300 packets while a few nodes send more than 1300 packets.

The power cost function has the smallest box and the shortest whiskers which indicates how closely packets are distributed among nodes. The large majority of nodes each send about 800 packets with an exception of a few outliers. This explains why this cost function yields the longest network lifetime, the traffic load is evenly shared among many nodes instead of burning a few nodes quickly.

Another interesting fact in Figure 29 is the location of the median (red line across the box). Unlike the other three cost functions, the median of the static routing is at the bottom of the box at about 300 packets. That means that half of nodes send only about 300 packets or less while the other half send anywhere between 300 and 1400 packets; again, another proof of the uneven load distribution. This point is further emphasized in Table 5 where we see a large difference between the mean and median of the packets distribution for the static routing compared to the other three cost functions. The lower median indicates that the majority of nodes send less packets; half of them sent less than 298. The gap between the mean and median indicates a widely spread distribution of packets as we saw in Figure 29.

*Table 5: Mean and Median of packets distribution among nodes for all 4 cost types*

| | **Average** | **Median** |
|---|---|---|
| Static | 509 | 298 |
| Exponential | 799 | 856 |
| Power | 781 | 823 |
| Ratio | 726 | 756 |

### 5.2 Effects of route restrictions

In the energy aware routing study in [10], nodes only forward packets to neighbors that are closer to the destination. This keeps nodes from sending packets backwards and packets taking longer routes than needed. We applied the same idea in our network to see the effects of restricting nodes to only using shorter routes. To do that we added another condition in the route selection process where a node saves only routes with less or equal number of hops (length) to its own. That means nodes cannot send data via a neighbor that is further away from the hub than themselves.

The results were rather surprising. When nodes only send packets to neighbors closer to the destination (Figure 30), the network lifetime is shorter than when there are no restrictions on route selection (Figure 31).

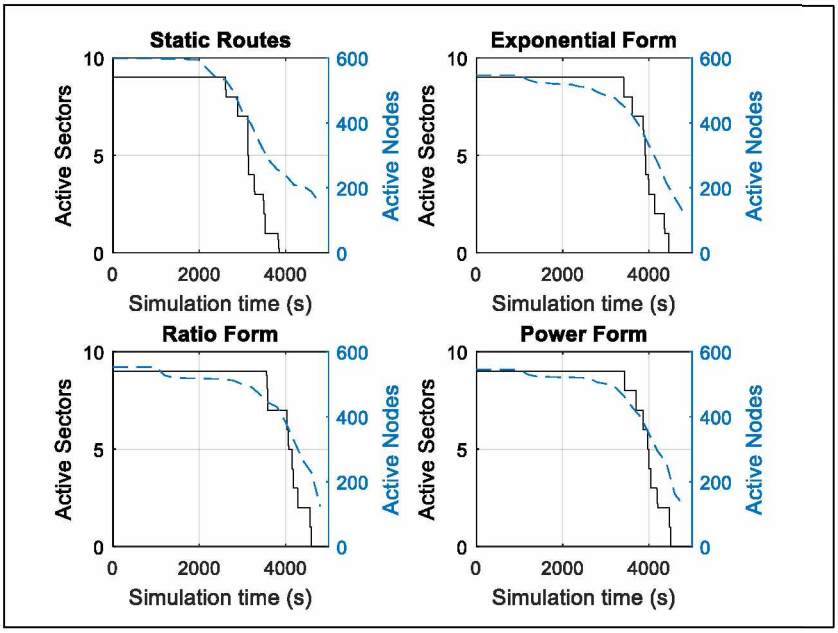



*Figure 30: Network lifetime performance with route restrictions*

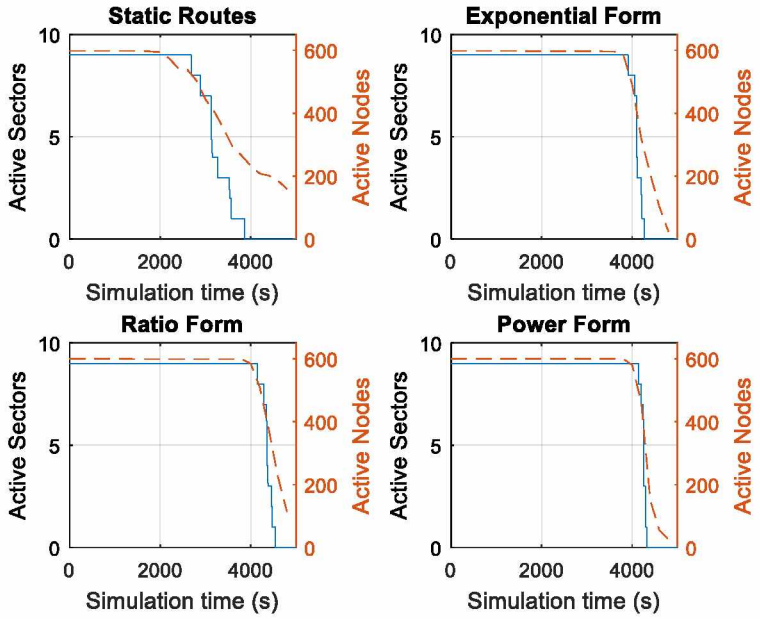



*Figure 31: Network lifetime performance without route restrictions*

This, at a first glance surprising discovery, can be explained by the fact that load balancing routing protocols benefit from having multiple routes available. Putting restrictions on routes such as using only shorter routes [10] limits the number of available routes for each node which diminishes the effects of load balancing.

The concern of nodes selecting unnecessary long routes is addressed by the idea of aggregate cost. For example, if a node has two routes, a short one (2 hops) and a long one (4 hops), the cost of each route is the sum of communication costs of nodes along each route. In normal conditions, the node will select the shorter route since it has the lowest cost. The exception comes when the shorter route has a higher cost meaning either the two nodes along that route are in critical conditions or they have poor links in which case they should be avoided anyways.

### 5.3 Multiple seed random deployment

In Castalia, random processes use the same seed unless the user chooses a different one. That means that when the network simulation is ran multiple times nodes end up in the same position every time. This is a very useful feature when trying to evaluate different solutions under the same conditions. However, we also wanted to evaluate the robustness of our routing protocol and to see what the performance of our cost functions would be for various random deployment. To do that, we employed Castalia's *–r N* command which runs the simulation *N* times each with a different random seed, as explained on page 36 of the User's Manual [41]. As an example, each of the four cost functions was ran with four different random seeds and the results are shown in Figure 32.

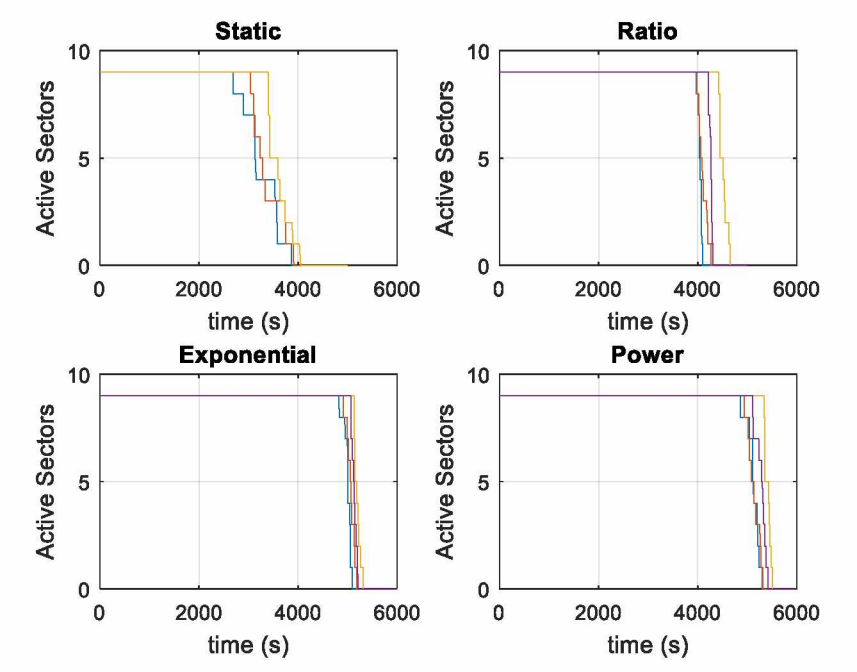



*Figure 32: Cost function performance for various random deployments*

Each graph (line) in any of the four plots in Figure 32 represents the lifetime performance for one random seed. Although each seed yields different results, the lifetime performance is fairly consistent

for all four cost functions. The exponential cost function is the most consistent. Also, the order of performance was not changed with power and exponential cost functions leading the way. This experiment shows that our routing protocol is likely independent of any given deployment and it works for any random deployment.

### 5.4 Link Quality and transmission efficiency

In this section, we evaluate how link quality (LQ) inside the cost function affects the lifetime performance of a network. As mentioned before, link quality is often not used in the routing decision process as we see in other routing protocol studies [10], [11], and [33], which can be justified by the fact that in those studies, the authors assume the link quality to be generally high for the entire network in which case the LQ would not make much difference. That is not the case in farming applications; LQ can vary tremendously due to obstacles such as trees, landscape, or farming machinery. Thus, in this section, we study how using the LQ in the cost function affects the network lifetime.

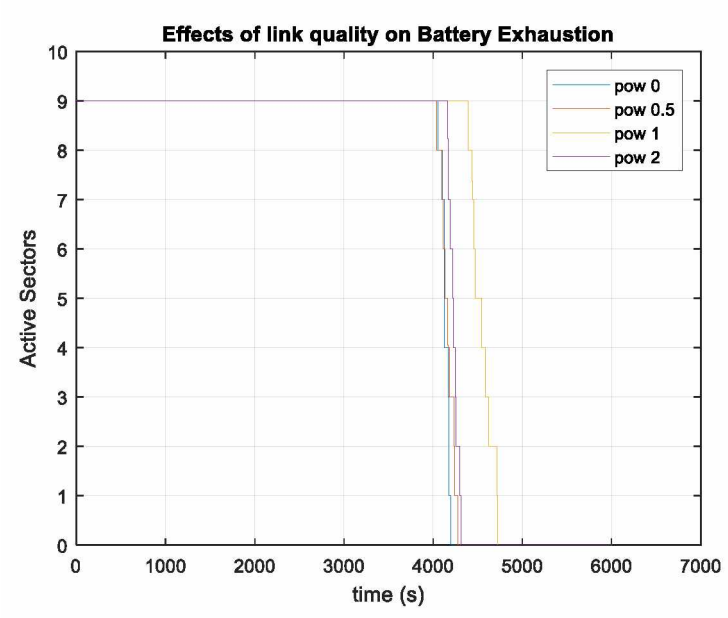
As mentioned in section 4.5.2, nodes store information about routes to their neighbors and each route entry in the table has the total packets and successful packets. These two fields are used to compute the link quality of each route using Equation 7 in section 4.6.2.

We incorporated the link quality into the cost function to allow nodes to choose not only the neighbor in good energy conditions but also one with the best link. There are many ways to include LQ in the cost function but in this study we used three formulas, and for each formula, we recorded the network lifetime for 4 different weights of LQ [0, 0.5, 1, 2]. These weights represent the exponent of LQ in the formula which is used to determine the importance of LQ in the selection process; from very significant (power of 2) to meaningless (power of 0).

In each formula, $C_i$ is the energy-based cost of the transmitting node. $C_j$ is the cost of lowest-cost neighbor, $LQ$ is the link quality of the link i-j. $pow$ is the weight parameter for LQ. Also note that the cost is inversely proportional to LQ, which means that a poor link quality leads to a high cost.

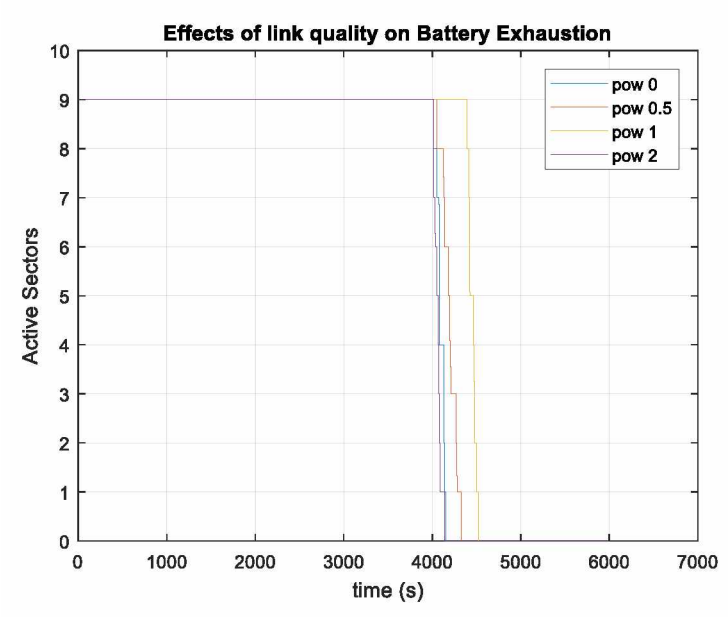
**Formula 1:**

$$\boldsymbol{C_{LQ} = C_i + C_j(\frac{1}{LQ})^{pow}} \quad (10)$$



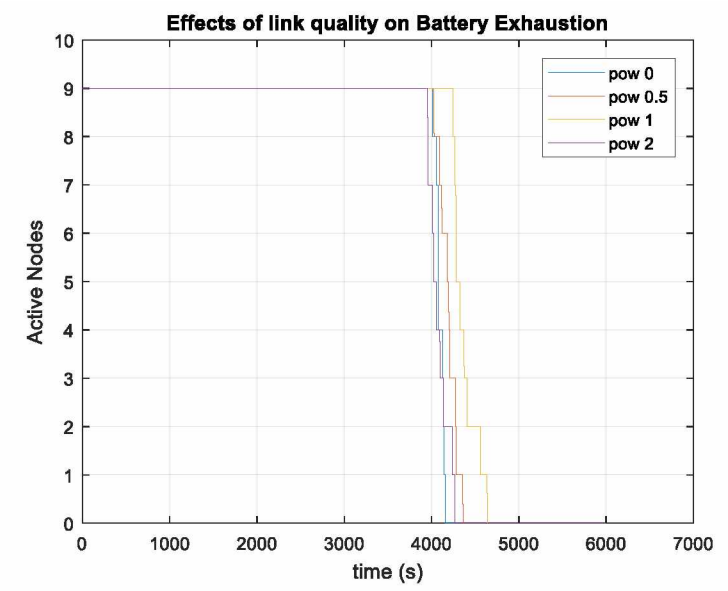

**Formula 2:**

$$C_{LQ} = C_i(\frac{1}{LQ})^{pow} + C_j \quad (11)$$




**Formula 3:**

$$C_{LQ} = (C_j + C_i)(\frac{1}{LQ})^{pow} \quad (12)$$




For each formula, we notice that using LQ improves the network lifetime, with power of 1 giving the best performance in all three cases. This confirms our prediction that adding LQ to the cost formula improves the network lifetime. We also notice that more weight for LQ is not necessarily good; power of 2 performs worse than power of 1. In fact, the performance of power of 2 is similar to that of power of 0. That is because giving more weight to LQ can overshadow the remaining energy effects forcing nodes to choose high LQ routes regardless of their battery state. It might be possible to obtain a better performance from a decimal weight between 1 and 2 but the user has to consider the capacity of the microcontroller to handle complex tasks such as floating point operations. Some small microcontroller are not equipped with a floating point unit [59] and might have to spend significant time performing these fractional powers.

To further understand how LQ affects the network, we recorded the retransmissions for each weight in all three formulas, as shown in Table 6 and Figure 33.

*Table 6: Packet retransmission for various powers of LQ in the 3 cost function formulas*

| LQ power | | 0 | 0.5 | 1 | 2 |
|---|---|---|---|---|---|
| Retrans./ 100 packets | Formula 1 | 24.3 | 16.9 | 14.2 | 13.2 |
| | Formula 2 | 22.5 | 22 | 18.3 | 15.2 |
| | Formula 3 | 22.5 | 17.1 | 13.5 | 13.1 |

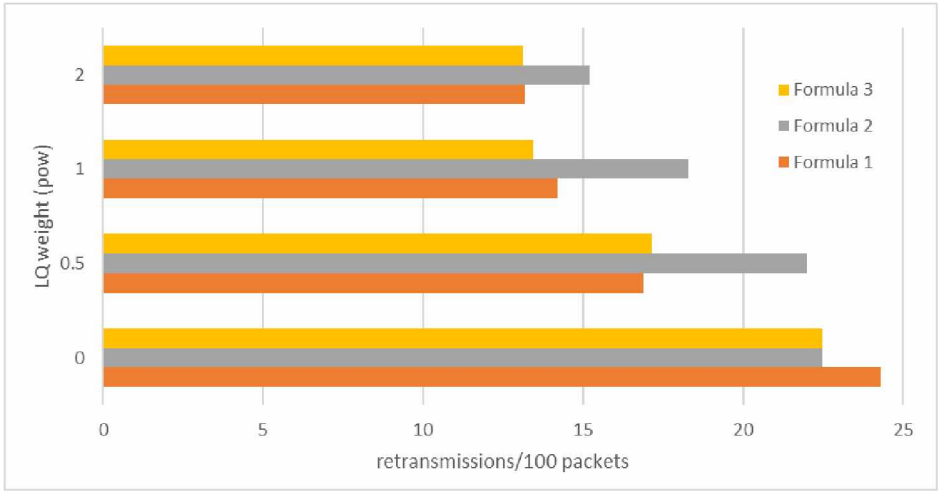



*Figure 33: LQ effects on retransmission*

In each of the three cases, using LQ reduces the number of retransmissions, even more so as the weight of LQ (power) increases. In Formula 1 for example, using a weight of 2 reduces the average number of retransmissions by 45%. This is what one would expect since nodes are favoring routes with higher link quality and therefore are less likely to drop packets.

As we mentioned before, considering the link quality is important because it can be very unpredictable in a farming environment. Our results show some improvement from using the LQ but the impact would be even more significant if we could emulate the obstacles in a farm which was not feasible in the simulation tool we used. Figure 34 shows the link quality status in the entire network and as we can see the majority of links are good: 60% of links have a LQ of 80% or more while only 15% have less than 50% quality. Those numbers, in our opinion, do not accurately reflect what would happen in a real farm and that reduces the effects of LQ in the cost function.

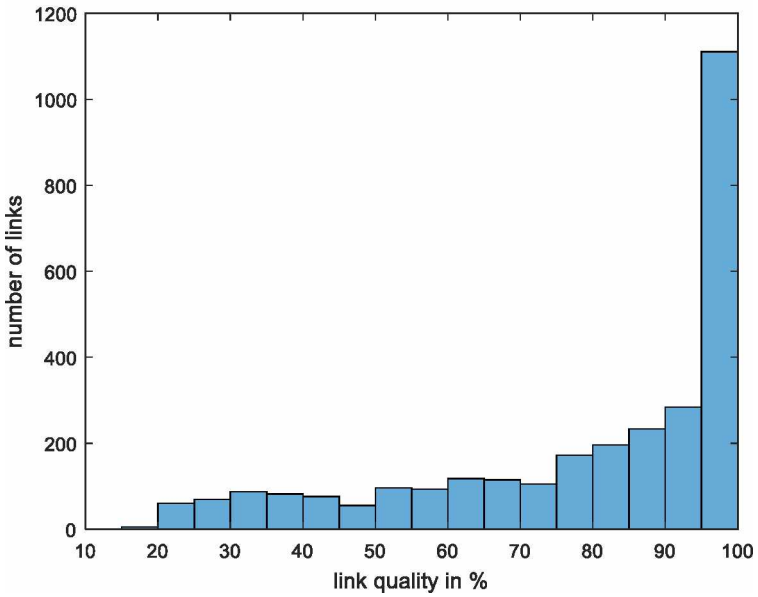



*Figure 34: Link quality distribution in network*

### 5.5 Extended simulations

So far, our simulations have been relatively short, about 6000 s long or less, and that is mainly because of the capacity limitations of the simulator and time constraints. Long simulations are computationally more complex and they take longer to execute.

Nonetheless, we wanted to see how the performance of our routing protocol and cost functions scale when the network is simulated for a longer period. To do that we doubled the nodes' battery capacity, from 300 mAs to 600 mAs, and increased the simulation time to 12,000 s. Then we ran the simulation with two sampling periods (20 s and 40 s). This parameter determines how often new packets are sent. The resulting network profiles are plotted in Figure 35.

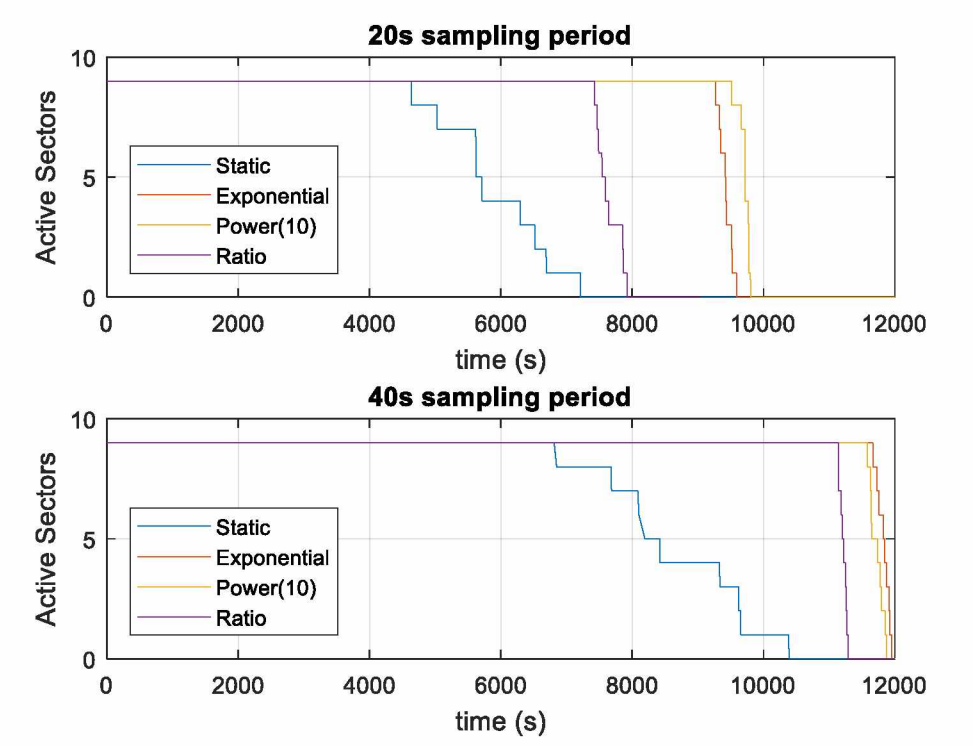



*Figure 35: Extend simulation of network lifetime performance*

As expected, the network lifetime is extended significantly compared to the previous short simulations. Another observation is that the gap between the performance of static routing (blue) and dynamic routing (red, orange, purple) is accentuated. The longer the simulation runs the more we see the effects of load balancing routing.

The two network simulations in Figure 35, though they started with the same amount of initial energy, do not behave the same. The 40 s sampling period has a longer lifetime for all 4 routing types. As we see in Table 6 – percentage of energy consumed by the radio, when the sampling period is increased from 20 s to 40 s, the radio activity is reduced for all 4 routing types because there is less packets being sent and as a result, the network lasts longer.

*Table 7: Energy consumed by radio (%) for 20 s and 40 s sampling period*

| Sampling period (s) | Static | Exponential | Power | Ratio |
|---|---|---|---|---|
| 20 | 30 | 49 | 48.2 | 46.1 |
| 40 | 20.1 | 34.6 | 31.7 | 27.2 |

### 5.6 Effects of cost function on packet delivery rate (PDR)

Our results so far have shown that using the cost function preserves the network lifetime and some functions such as power can increase the network lifetime even more. This benefit however does not come without a cost; energy efficient cost functions reduce the packet delivery rate (PDR). Packet delivery was not the primary focus of our study, but it is an important factor to consider in developing routing protocols since saving energy might be achieved at the cost of losing packets as we see in our case.

To see the how the cost function affects the PDR, we ran the network simulation with each one of the four cost functions for a period of 2000 s – the results are summarized in Table 7. The delivery rate is generally higher for dynamic routing (ratio, exponential, and power) compared to static routing.

*Table 8: Packet delivery rate for the 4 routing cost functions*

| | Static | Ratio | Exponential | Power |
|---|---|---|---|---|
| **Sent packets** | 27,533 | 27,569 | 27,565 | 27,527 |
| **Received packets (hub)** | 22,950 | 25,894 | 25,316 | 23,720 |
| **Packet delivery rate** | *83.3%* | *93.9%* | *91.8%* | *86.1%* |

But perhaps the most interesting observation from Table 7 is that for the 3 dynamic routing types, the ratio cost function has the highest PDR, followed by the exponential and the power in that order; what is interesting is that they come in reverse order of the lifetime performance as we saw in section 5.1. Load-balancing routing saves energy by spreading packets across the network instead of using a few routes. As packets spread across the network looking for routes to the hub, their likelihood to fail increases. That explains why a more energy-efficient cost function such as power, has a lower PDR.

This inverse relationship between PDR and energy saving was also observed within each cost function. The PDR of the exponential cost function, $Cost = A * e^{(B\frac{E_i}{R_i})}$, drops as the value of the parameter $B$ increases, as seen in Table 8. The same thing happens for the power function $Cost = (\frac{E_i}{R_i})^p$, the PDR decreases as the exponent $p$ rises, as shown in Table 9. Yet, we know that higher values of $p$ yield better lifetime performance [48].

*Table 9: Packet delivery rate for different values of B in exponential cost function*

| Value of $B$ in $A * e^{(B\frac{E_i}{R_i})}$ | 2 | 5 | 10 | 20 | 50 |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| PDR | 92.6% | 91.8% | 90.1% | 71.4% | 49.4% |

*Table 10: Packet delivery rate for different values of p in power cost function*

| Value of $p$ in $Cost = (\frac{E_i}{R_i})^p$ | 2 | 5 | 10 | 25 | 50 |
|---|---|---|---|---|---|
| PDR | 92.6% | 90.1% | 86.1% | 59.1% | 44.7% |

The results of this experiment have shown that there is a tradeoff between saving energy and increasing the packet delivery rate. It is up to the user to determine what they value the most in their network, extended lifetime or high packet reception rate, and then chose the cost function accordingly. Typically, farming applications do not require high sampling rate; for example, Parkin describes a study on soil sampling frequency where rainfall, soil water content, and temperature were sampled at an hourly rate [60]. With a relatively higher sampling rate, farming WSN can afford to lose some packets. That however may vary depending on what is being sampled. But luckily, the user has the option to choose a cost function that meets the requirements.

# Chapter 6 Conclusion and future improvements

In this research project, we studied the ability of load balancing routing protocols to extend the lifetime of a precision agriculture WSN by managing the energy resource more efficiently. Using three types of functions – ratio, power, and exponential, we evaluated the effects of cost function on lifetime performance and we compare dynamic routing in general to static routing. Simulation results show that dynamic routing protocols (ratio, power, exponential) preserve the network longer than static routing. By carefully studying what happens at the packet traffic level, we were able to understand how load-balancing preserves the network by dynamically choosing healthier routes and avoiding at-risk nodes. First, we observed that in dynamic routing (using cost function) the large majority of nodes are involved in packet transmission. While in static routing only a few nodes are involved in packet routing. We also noticed that the packet load is unevenly distributed among nodes in static routing with 50% of the nodes carrying as little as 300 packets, while a few other nodes send about 1400 packets. In load balancing however, traffic was evenly shared among nodes. Based on the cost function used, dynamic protocols performed differently with the power function exhibiting the best performance in terms of network lifetime. The benefit of longer network lifetime nonetheless comes with a cost and that is packet delivery rate. We observed that the most energy-efficient cost functions, such as power, have lower packet delivery rates. Hence, the user needs to consider their packet delivery in the choice of cost function.

Furthermore, we explore the impact of link quality, another factor that affects the network. We found that in a network with unbalanced and unpredictable link quality, such as those found in an agricultural environment, adding link quality to the cost function improves the lifetime performance. We also predicted that with many links having poor quality, a scenario that we were not able to create due to our simulator's limitations, the link quality effects on lifetime would be more significant.

Another major contribution of this study is the creation of a platform for testing routing protocols. In order to test our own routing protocol, we built a wireless sensor network simulation in Castalia, thoroughly described in Chapter 4. Following Castalia's highly parametric nature, our simulation was designed to be very flexible and parametric. For example, the implementation of the cost functions was done using a switch-case statement, as seen in Figure 36, and thus new cost functions can be implemented by adding more cases to the switch statement. The coefficients in each cost function are also variables that the user can easily modify as needed.

```
void TwoLayerNetwork::update_com_cost(){

    if(batteryLeft > 0 && myRoutes.size() > 0)
    {
        minNeighborCost();
        route minRoute = myRoutes.at(0);
        switch(routeType)
        {
            case 1:
            {
                com_cost = cost_varA*exp(-cost_varB*batteryLeft/E_initial);
                break;
            }
            case 2:
            {
                com_cost = pow(E_initial/batteryLeft, cost_pow);
                break;
            }
            case 3:
            {
                com_cost = (E_initial/batteryLeft);
                break;
            }
        }

        com_cost += minRoute.cost;

    }

}
```

*Figure 36: Switch-case implementation of the cost functions*

Moreover, other network parameters such as the sampling period and initial battery capacity are user inputs that the user can easily set or change in the configuration file. The simulation was intentionally designed to be open and easily modifiable to allow other researchers to build on it, to improve it, or to conduct new routing studies. From our own experience in this study, building the simulation test bed took a good amount of time and effort that could have been dedicated to the studying the results.

As for improvements, there are several areas that can be further studied. New types of cost functions that use more factors, such as neighbors, can be evaluated. It would also be interesting to study the performance of load-balancing in terms of other network qualities, such as throughput and latency. From this study and other similar studies, it is very clear that load-balancing routing improves the network lifetime considerably, but there is not enough study on how cost-based routing protocols affect the packet throughput and latency. Packets in this type of network take long paths which increases latency and the likelihood of packet failure. More research can be done on how load balancing routing interacts with other network layers, in particular the MAC layer. Most routing studies do not address medium access but MAC protocols can have a significant impact on the performance of the routing protocol they are used with.

# References


[1] T. Instruments, "MSP430 ultra-low-power sensing & measurement MCUs," Texas Instruments, [Online]. Available: http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html. [Accessed 24 05 2018].

[2] N. Wang, N. Zhang and M. Wang, "Wireless sensors in agriculture and food industry - Recent development and future perspective," *Computers and Electronics in Agriculture,* vol. 50, pp. 1-14, 2006.

[3] USDA, "Ag and Food Sectors and the Economy," United States Department of Agriculture, 18 October 2017. [Online]. Available: https://www.ers.usda.gov/data-products/ag-and-food-statistics-charting-the-essentials/ag-and-food-sectors-and-the-economy/. [Accessed 22 February 2018].

[4] United States Department of Agriculture, "Agriculture Technology," National Institute of Food and Agriculture , [Online]. Available: https://nifa.usda.gov/topic/agriculture-technology. [Accessed 10 04 2018].

[5] K. Langendoen, A. Baggio and O. Visser, "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture," in *Parallel and Distributed Processing Symposium*, Rhodes Island, 2006.

[6] A.-J. Garcia-Sanchez, F. Garcia-Sanchez and J. Garcia-Haro, "Wireless sensor network deployment for integrating video surveillance and data-monitoring in precision agriculture over distributed crops," *Computers and Electronics in Agriculture,* no. 75, pp. 288-303, 2011.

[7] J. Riquelme, F. Soto, J. Suardiaz, P. Sanchez, A. Iborra and J. Vera, "Wireless Sensor Networks for precision horticulture in Southern Spain," *Computers and Electronics in Agriculture,* no. 68, pp. 25-35, 2009.

[8] MEMSIC, "Micaz Datasheet," [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf. [Accessed 22 February 2018].

[9] B. Krishnamachari, Network Wireless Sensors, New York: Cambridge University Press, 2005.

[10] R. C. Shah and J. M. Rabaey, "Energy Aware Routing for Low Energy Ad Hoc Sensor Networks," *2002 IEEE Wireless Communications and Networking Conference Record. WCNC 2002 (Cat. No.02TH8609),* pp. 350 - 355, 2002.

[11] A. Liu, J. Ren, X. Li, Z. Chen and X. Shen, "DCFR: A novel Double Cost Function based Routing algorithm for wireless sensor networks," *IEEE International Conference on Communications (ICC),* pp. 323-327, 2012.

[12] N. Zhang, M. Wang and N. Wang, "Precision agriculture*/a worldwide overview," *Computers and Electronics in Agriculture,* vol. 36, pp. 113-132, 2002.

[13] G. C. Rains and D. L. Thomas, "Precision Farming an Introduction," March 2009. [Online]. Available: https://pdfs.semanticscholar.org/671d/74885acc484fcb4d9dcb04e95b8395e5131b.pdf. [Accessed 28 December 2017].

[14] S. Daberkow and W. McBride, "Adoption of precision agriculture technologies by US farmers," in *Proceedings of Fifth International Conference on Precision Agriculture*, Bloomington, 2000.

[15] J. Popp and T. Griffin, "Adoption trends of early adopters of precision farming in Arkansas," in *Proceedings of Fifth International Conference on Precision Agriculture*, Bloomington, 2000.

[16] G. M. Insight, "PRECISION FARMING MARKET SIZE BY COMPONENT," Global Market Insight, 02 2018. [Online]. Available: https://www.gminsights.com/industry-analysis/precision-farming-market?utm_source=globenewswire.com&utm_medium=referral&utm_campaign=Paid_globenewswire. [Accessed 11 04 2018].

[17] AGCO, "Who we are," AGCO, 2018. [Online]. Available: http://www.agcocorp.com/about.html. [Accessed 11 04 2018].

[18] Agribotix, "Agribotix home page," Agribotix, [Online]. Available: https://agribotix.com/. [Accessed 11 04 2018].

[19] AgSense, "AgSense home page," AgSense LLC, [Online]. Available: http://www.agsense.com/. [Accessed 11 04 2018].

[20] T. Atlas, "The average cost of IoT sensors is falling," Atlas, 2014. [Online]. Available: https://www.theatlas.com/charts/BJsmCFAl. [Accessed 04 05 2018].

[21] T. Tzach, "Soil Sensors: A New Direction in Precision Agriculture to Improve Crop Production," Precision Ag, 10 04 2018. [Online]. Available: http://www.precisionag.com/systems-management/soil-sensors-a-new-direction-in-precision-agriculture-to-improve-crop-production/. [Accessed 11 04 2018].

[22] Aqeel-ur-Rehman, A. Z. Abbasi, N. Islam b and Z. A. Shaikh b, "A review of wireless sensors and networks' applications in agriculture," *Computer Standards & Interfaces,* vol. 36, pp. 263-270, 2014.

[23] http://www.precisionag.com/systems-management/soil-sensors-a-new-direction-in-precision-agriculture-to-improve-crop-production/, "Mahlein, Anne-Katrin," *Plant Disease,* vol. 100, no. 2, pp. 241-251, 2016.

[24] M. Perkins, N. Correal and B. O'Dea, "Emergent wireless sensor network limitations: a plea for advancement in core technologies," in *IEEE Sensors*, Orlando, FL, 2002.

[25] M. Damas, A. Prados, F. Gomez and G. Olivares, "HidroBus system: fieldbus for integrated management of extensive areas of irrigated land," *Microprocessors Microsyst,* vol. 25, pp. 177-184, 2001.

[26] W. R. Heinzelman, A. Chandrakasan and H. Balakrishnan, "Energy-Efficient Communication Protocol for Wireless Microsensor Networks," *Proceedings of the Hawaii International Conference on System Sciences,* vol. 2, p. 10, 2000.

[27] S. D. MURUGANATHAN, D. C. F. MA, R. I. BHASIN and A. O. FAPOJUWO, "A Centralized Energy-Efficient Routing Protocol for Wireless Sensor Networks," in *IEEE Radio Communications* , 2015.

[28] W. B. Heinzelman, A. P. Chandrakasan and H. Balakrishnan, "An application-specific protocol architecture for wireless microsensor networks," *IEEE Transactions on Wireless Communications,* vol. 1, no. 4, pp. 660 - 670, 2002.

[29] S. Lindsey, C. Raghavendra and K. M. Sivalingam, "Data gathering algorithms in sensor networks using energy metrics," *IEEE Transactions on Parallel and Distributed Systems,* vol. 13, no. 9, pp. 924 - 935, 2002.

[30] F. Farazandeh, R. Abrishambaf, S. Uysa, T. Gomes and J. Cabral, "A Hybrid Energy-Efficient Routing protocol for Wireless Sensor Networks," in *IEEE International Conference on Industrial Informatics (INDIN)*, Bochum, Germany, 2013.

[31] S. Singh, M. Woo and C. S. Raghavendra, "Power-Aware Routing in Mobile Ad Hoc Networks," in *Proceedings of ACM MobiCom*, 1998.

[32] R. C. Shah and J. M. Rabaey, "Energy aware routing for low energy ad hoc sensor networks," in *Wireless Communications and Networking Conference*, Orlando, 2002.

[33] C.-S. Ok, S. Lee, P. Mitra and S. Kumara, "Distributed energy balanced routing for wireless sensor networks," *Computers & Industrial Engineering,* vol. 57, pp. 125-135, 2009.

[34] A. Rogers, E. David and N. R. Jennings, "Self-organized routing for wireless microsensor networks," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans,* vol. 35, no. 3, pp. 349-359, 2005.

[35] M. Ettus, "System capacity, latency, and power consumption in multihop-routed SS-CDMA wireless networks," in *Radio and Wireless Conference, 1998. RAWCON 98. 1998 IEEE*, Colorado Springs, 1998.

[36] H. L. V. K. D. M. A. &. P. B. Harsh Sundani, "Wireless Sensor Network Simulators," *International Journal Of Computer Networks,* vol. 2, no. 5, pp. 249-256, 2011.

[37] E. Egea-Lopez, J. Vales-Alonso, A. Martinez-Sala, P. Pavon-Marino and J. Garcia-Haro, "Simulation Tools for wireless Sensor Networks," in *Summer Simulation Multiconference - SPECTS*, 2005.

[38] M. Korkalainen, M. Sallinen, N. Kärkkäinen and P. Tukeva, "Survey of Wireless Sensor Networks Simulation Tools for Demanding Applications," in *2009 Fifth International Conference on Networking and Services*, 2009.

[39] M. P. Chhimwal, D. Singh Rai and R. Deepesh, "Comparison between Different Wireless Sensor Simulation Tools," *Journal of Electronics and Communication Engineering,* vol. 5, no. 2, pp. 54-60, 2013.

[40] "Castalia Wireless Sensor Simulator," [Online]. Available: https://castalia.forge.nicta.com.au/index.php/en/. [Accessed 24 January 2017].

[41] A. Boulis, *Castalia User's Manual,* 2011.

[42] A. Boulis, "Castalia Simulator Forum," NICTA, [Online]. Available: https://groups.google.com/forum/#!forum/castalia-simulator.

[43] OpenSim, "Omnet++," OpenSim, [Online]. Available: https://www.omnetpp.org/. [Accessed 11 12 2017].

[44] M. Zuniga and B. Krishnamachari, "Analyzing the Transitional Region in Low Power Wireless Links," in *First Annual IEEE Communications Society Conference on*, Santa Clara, 2004.

[45] B. Krishnamachari, Networking Wireless Sensors, New York: Cambridge University Press, 2005.

[46] T. Instruments, "TI CC2420 Datasheet," [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2420.pdf. [Accessed 05 12 2017].

[47] A. Camilli, C. E. Cugnasca and S. A. M., "From wireless sensors to field mapping: Anatomy of an application for precision agriculture," *Computers and Electronics in Agriculture,* vol. 58, pp. 25-36, 2007.

[48] J.-H. Chang and L. Tassiulas, " Energy conserving routing in wireless ad-hoc networks," *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies,* vol. I, pp. 22-31, 2000.

[49] T. Instruments, "TI MSP430F5438A Datasheet," [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430f5438a.pdf. [Accessed 08 12 2017].

[50] Wikipedia, "AA battery," Wikipedia, 28 11 2017. [Online]. Available: https://en.wikipedia.org/wiki/AA_battery. [Accessed 8 12 2017].

[51] S. Park, A. Savvides and M. B. Srivastava, "Battery capacity measurement and analysis using lithium coin cell battery," in *Low Power Electronics and Design*, Huntington Beach, 2001.

[52] L. M. Feeney, C. Rohner and A. Lindgren, "How do the dynamics of battery discharge affect," in *Wireless On-demand Network Systems and Services (WONS)*, Obergurgl, 2014.

[53] J. Riquelme, F. Soto, J. Suardíaz, P. Sánchez, A. Iborra and J. Verab, "Wireless Sensor Networks for precision horticulture in Southern Spain," *Computers and Electronics in Agriculture,* vol. 68, no. 1, pp. 25-35, 2009.

[54] M. Prinn, L. Moore, M. Hayes and B. O'Flynn, "Comparing Low Power Listening Techniques with," in *International Conference on Smart Systems, Devices and Technologies*, Paris, 2014.

[55] M. Magno and L. Benini, "An Ultra Low Power High Sensitivity Wake-Up," in *International Workshop on GReen Optimized Wireless Networks*, Larnaca, Cyprus, 2014.

[56] A. M. (AMS), "AS3933 LF Receiver IC," Austria Microsystems (AMS), [Online]. Available: http://ams.com/eng/Products/Wireless-Connectivity/Wireless-Sensor-Connectivity/AS3933. [Accessed 05 12 2017].

[57] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *international conference on Embedded networked sensor systems*, Los Angeles, 2003.

[58] Y. Tselishchev, A. Boulis and L. Libman, "Experiences and Lessons from Implementing a Wireless Sensor Network MAC Protocol in the Castalia Simulator," in *IEEE Wireless Communications and Networking Conference*, Sydney, 2010.

[59] Wikipedia, "Floating Point Unit," Wikipedia, 18 03 2018. [Online]. Available: https://en.wikipedia.org/wiki/Floating-point_unit. [Accessed 18 04 2018].

[60] T. B. Parkin, "Eff ect of Sampling Frequency on Estimates of Cumulative Nitrous Oxide Emissions," *Journal of Environmental Quality,* vol. 37, pp. 1390-1395, 2008.

# Appendix

## Appendix A

### A.1 Collision and channel models in Castalia

Castalia offers different types of collision and channel models. One of those is the naïve model also called the disc model. The naïve model in Castalia is a simplistic model of the wireless channel that removes the randomness of the channel and packet collisions. It offers ideal conditions in wireless communication where any node within the transmission range of a received is guaranteed to receive the packets. In our simulations we used collision model 2 described further down, because it offers more realistic behavior. Though not realistic, the naïve model can be very useful in testing new routing algorithm in an ideal environment. Channel randomness and packet collisions are often unpredictable yet they can heavily affect the performance of routing protocols, so the user might find it useful to isolate them for testing purposes.

To understand the naïve model further, let us briefly describe how packet transmission is handled in Castalia. When a node needs to send a packet, its radio module sends a message to the wireless channel (WC) which represents the wireless communication medium. The WC then sends a message to the receiving node with the relevant information about the signal namely the bandwidth, the modulation, and most importantly the signal strength in dBm. The WC calculates the signal strength at the receiver using the transmission power that was used to send the packet and the path loss (average path loss and temporal variation). Upon receiving a signal from the WC, the radio of the receiving node computes the signal to interference plus noise ratio (SINR) and depending on the sensitivity, it determines whether the packet is successful or not. SINR calculation is particularly important because at any given time, there might be more than one signal reaching the radio. Signal delivery is thoroughly described on *pg. 59* of the user manual for Castalia [41].

Going back to the naïve model, we have understood that signal strength is the primary factor in determining the success of packet reception. Signal strength is affect by path loss incurred during propagation and the average path loss in Castalia is calculated using the lognormal shadowing model [44] which has a Gaussian zero-mean random variable (Xσ):

$$PL(d) = PL(\mathrm{d0}) + 10 \cdot \eta . \log\left(\frac{d}{d0}\right) + X\sigma .$$

The naïve model removes the random variable from the path loss and as a result, nodes located at the same distance from a transmitter receive the exact same signal strength. In addition, all the links are bidirectional, which means that the quality of link A->B is the same as the quality of link B->A. In the naïve model, the user can choose the *IDEAL* modulation scheme which, together with the zero-randomness, results in a simple unit disk model. The disk represents a range around the node within which every neighbor can hear from the node.

Another important feature of the naïve model is the ability to choose the communication range by adjusting the PL(d0) parameter of the wireless channel. And lastly, the user can choose a collision free model for the radio.

In order to test the naïve model, we used a simple algorithm to find neighbors. The setup of our experiment is a network of 100 nodes deployed in a 100 × 100 m field. The sink node is located at position (0, 0) and the other 99 nodes are randomly placed in the field. The transmission power was set to -10 dBm and the transmission range was set to 30 m.

The algorithm to find neighbors was designed as follows:

- After a randomly selected delay, each node broadcasts a FIND_NEIGHBOR packet. The random delay was used to allow nodes to listen while one of their neighbors is transmitting since nodes are not able to receive packets while in TX mode.
- When a node receives a FIND_NEIGHBOR packet, it responds with an ACK packet
- When the sender receives an ACK packet, it adds that neighbor to an array
- If a node has less than two neighbors, it continuously sends FIND_NEIGHBOR packets every 20 s or else every 40 s.
- The initial random delay is randomly chosen between 0 and 80 s

The simulation was run for 150 s and the resulting network is shown in Figure 37 below:

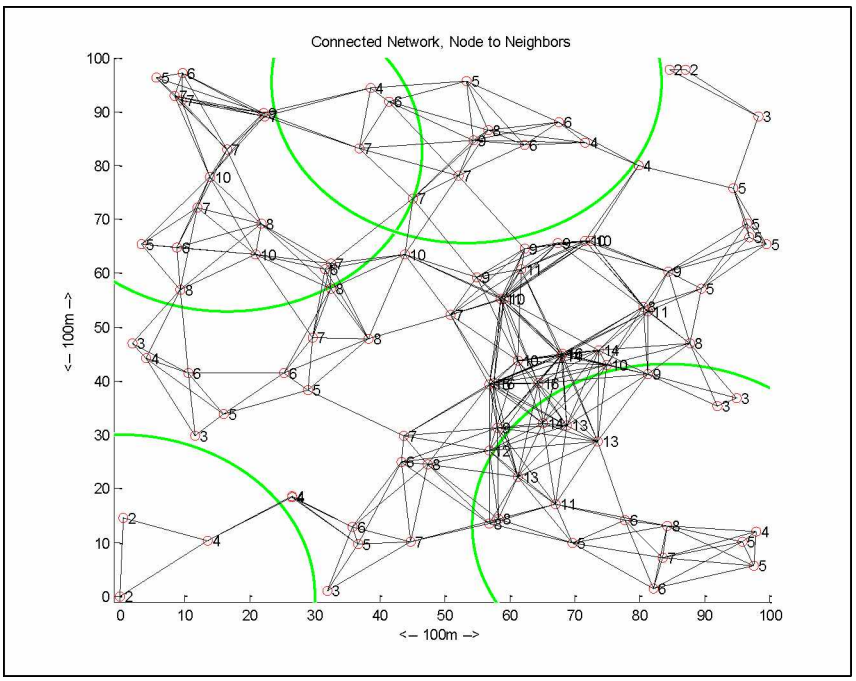



*Figure 37: Network connectedness for a naive model*

In this experiment, we used the average number of neighbors per node to measure the level of connectivity within the network. In the naïve model with no collisions (collision model 0) the average number of neighbors was 7.4 per node. For the sake of comparison, we ran the same experiment with non-idealistic network settings. More specifically, we enabled the randomness in the wireless channel and experimented with the other two collision models available in Castalia [41]: model 1 and 2. In collision model 1, whenever there is interference, packets cannot be received, while collision model 2 uses the signal to interference ratio (SIR) to determine whether a packet is received or not.

The resulting network is shown in Figure 38 and Figure 39 for collision model 1 and 2 respectively. We immediately noticed that both settings have less connections than the naïve model (Figure 37) and that is mostly due to collisions. Also, the length of links in the naïve model (Figure 37) is very consistent compared to the non-ideal environment (Figure 38 and Figure 39); that is the effect of randomness in the channel. A node might be able to communicate to a far-away neighbor while failing to talk to another neighbor much closer.

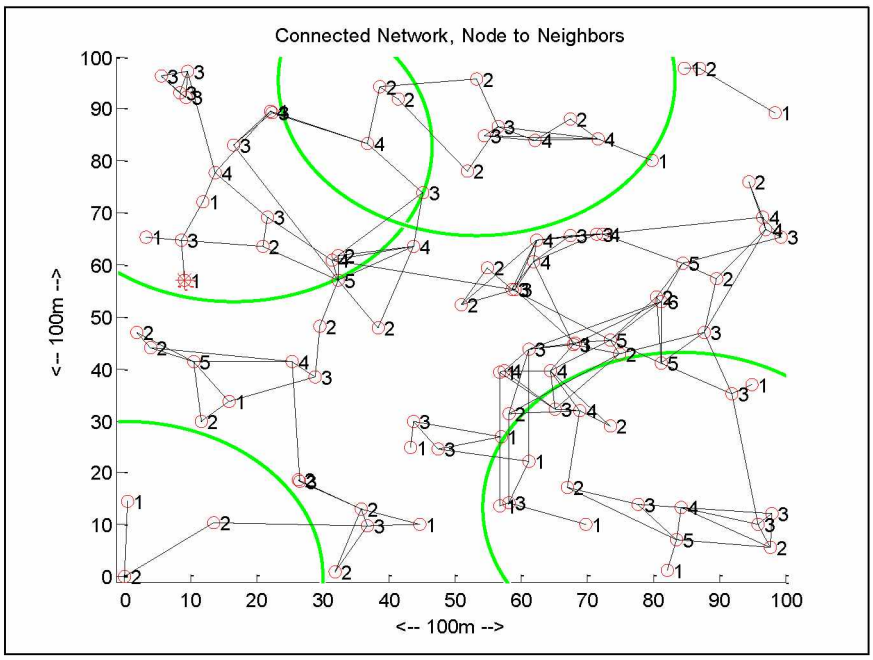



*Figure 38: Network connectedness for collision model 1*

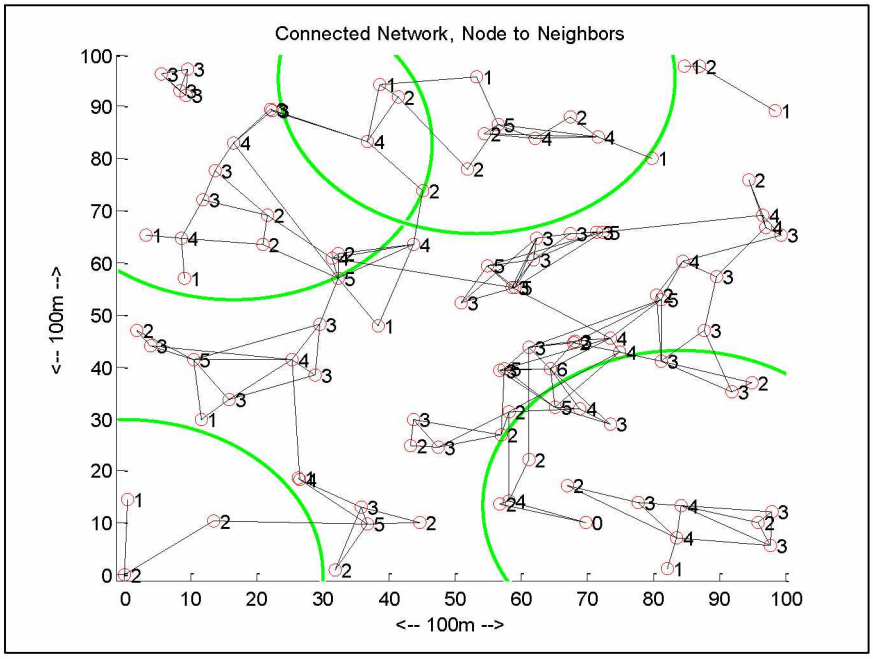



*Figure 39: Network connectedness for collision model 2*

Table 10 shows the average number of neighbors from the above experiment for collision model 1 and 2. As expected, there is less connectivity compared to the naïve model. For collision model 1, increasing the transmission power results in even less connectivity. That is because increasing TX power results in more interference and for collision model 1, any interference results in packet failure.

*Table 11: Average neighbors per node for collision model 1 & 2 at various transmit power*

| | -12dBm | -3dBm |
|---|---|---|
| Collision Model 1 | 2.68 | 1.31 |
| Collision Model 2 | 2.89 | 3.16 |

The naïve model in Castalia allows user to create ideal conditions for wireless communication where they can isolate certain factors such as channel randomness and collisions in order to test new algorithms. However, this model might not be suitable to evaluate the behavior of an algorithm in a real network and it should be used carefully.

## Appendix B: Data processing scripts

### B.1 Network visualization Matlab script

```
close all;
clear all;
clc;
% Network Parameters
x_lim = 200;
y_lim = 200;
Tx_pwr = -10;
Nodes = 600;
InitEnergy = 500;
max_time = 5000;
radius = (x_lim/100)*ones(Nodes,1);
range = 30*ones(Nodes,1);     % This radius represents the node's range
(30m) represented by a circle of that radius
NodeInfo = csvread('Node_Coordinates.csv',1,1);
neighbors = csvread('neighbors.csv');
Centers = NodeInfo(:,1:2);
neighbor = NodeInfo(:,3);
NeighborRSSI = NodeInfo(:,4);
Dist2Sink = NodeInfo(:,5);
energy = NodeInfo(:,6);
time_to_die = NodeInfo(:,8);
% Draw circles for each node with a given radius
%neighbor(1:12) = 0;
viscircles(Centers, 0.25*radius, 'EdgeColor', 'b');
viscircles(Centers(1:9,:), radius(1:9), 'EdgeColor', 'r');
% viscircles(Centers(energy == 0,:), 0.5*radius(energy == 0), 'EdgeColor',
'r');
xlim([-1 x_lim]);
ylim([-1 y_lim]);
%grid on;
hold on;
    % Label each node by its number
    % Draw link node-neighbor
    % Label link by # of attempts to connect
for i=1:Nodes
    str=sprintf(' %d',i-1);
%     if(time_to_die(i) ~= neighbor(i))
%         str = sprintf(' %d', i-1);
%         text(Centers(i,1), Centers(i,2), str);
%     end
    next_pt = neighbor(i);
    % Draw link between nodes and their cluster heads or direct neighbor
    if(next_pt >= 0)       % next_pt = -1 means node doesn't have a
cluster head
        x1 = Centers(i,1);
        x2 = Centers(next_pt+1,1);
        y1 = Centers(i,2);
        y2 = Centers(next_pt+1,2);
%         plot([x1 x2], [y1 y2], 'k-');       % Link node -> cluster
head/neighbor

    end
```

```
end
hold off;

title_str = sprintf('Sensor Deployment', Nodes);
title(title_str);
% xlabel(sprintf('<-- %dm -->', x_lim));
% ylabel(sprintf('<-- %dm -->', y_lim));

set(gca,'xtick',[0:x_lim/3:x_lim]);
set(gca,'ytick',[0:x_lim/3:x_lim]);
grid on;
%-------------------------------------------------------------------------
% The following code visualize the nodes and the connections to their
% neighbors i.e. those they heard from
% Disconnected nodes are marked with a red x
%-------------------------------------------------------------------------
figure;
viscircles(Centers, 0.25*radius, 'EdgeColor', 'b');
viscircles(Centers([0:8]+1, :), 0.5*radius([0:8]+1, :), 'EdgeColor', 'c');
% viscircles(Centers(Dist2Sink == 0,:), 0.5*radius(Dist2Sink == 0),
'EdgeColor', 'g');
hold on;
[row,col] = size(neighbors);



for r = 1:row
    n1 = neighbors(r,1);    % Origin node
    n2 = neighbors(r,3);        % End node

    plot([Centers(n1+1,1) Centers(n2+1,1)], [Centers(n1+1,2)
Centers(n2+1,2)], 'k:');
    if(neighbors(r,2) < 0.9)
        str = sprintf('%.0f', 100*neighbors(r,2));
%       text((Centers(n1+1,1)+ Centers(n2+1,1))/2, (Centers(n1+1,2)+
Centers(n2+1,2))/2, str);
    end
%     if(Dist2Sink(n1+1) < 0.8)
%         str = sprintf('%.0f', 100*Dist2Sink(n1+1));
%         text(Centers(n1+1,1), Centers(n1+1,2), str);
%     end
end
%Label Nodes with ID, Cost, Or any other interesting parameter
for m = 1:Nodes
    if(time_to_die(m) == 0)        % If Energy dropped to 5% or below
        viscircles(Centers(m,:), 0.25*radius(m), 'EdgeColor', 'r');
%     elseif(energy(m)/InitEnergy < .1)      % If Energy dropped to 30% or
below
%         viscircles(Centers(m,:), 0.25*radius(m), 'EdgeColor', 'y');
        str=sprintf(' %.0f',100*NeighborRSSI(m));
        text(Centers(m,1), Centers(m,2), str);
    end
end
hold off;
```

```
xlim([-1 x_lim]);
ylim([-1 y_lim]);
title(sprintf('Network Performance[Data Traffic] %ds Simulation red: dead,
y:<10%% energy', 10000));
% xlabel(sprintf('<-- %dm -->', x_lim));
% ylabel(sprintf('<-- %dm -->', y_lim));

set(gca,'xtick',[0:x_lim/3:x_lim]);
set(gca,'ytick',[0:x_lim/3:x_lim]);
grid on;

% Network performance
firstNodeDeadTime = min(time_to_die(time_to_die > -1));
fprintf('The first node dies after %.0fs\n', firstNodeDeadTime);
a = sort(time_to_die(time_to_die > -1));
Network_per = 0.1;
NetworkLifeTime = a(cast(Nodes*Network_per, 'uint8'));
fprintf('%d%% of the network dies after %.0fs\n', Network_per*100,
NetworkLifeTime);
deadNodes = length(time_to_die(time_to_die > -1))/length(time_to_die);
fprintf('%.0f%% of nodes are dead.\n', deadNodes*100);

% Ploting Active node vs. time
time = 1:200:max_time;
ActiveNode = [];
for i=1:length(time)
    ActiveNode = [ActiveNode 600-length(time_to_die(time_to_die >-1 &
time_to_die < time(i)))];
end

SectorNode = [];
for k = 0:8
    SectorNode = [SectorNode length(neighbor(neighbor == k))];
end
ActiveSectors = [];
for m=1:length(time_to_die)
    if(time_to_die(m) > 0)
        pos = neighbor(m);
        if(pos>=0)
            SectorNode(pos+1) = SectorNode(pos+1) - 1;
        end
    end
   ActiveSectors(m) = length(SectorNode(SectorNode > 40));
end

figure;
plot(sort(time_to_die), ActiveSectors, 'k-');
ylabel('Active Sectors');
ylim([0 10]);
hold on;
yyaxis right;
plot(time, ActiveNode, '--');
hold off;
ylabel('Active Nodes');
ylim([0 600]);
xlabel('Simulation time (s)');
```

```
grid on;
xlim([0 max_time]);
```

### B.2 Network lifetime plotting script

```
close all;
clear all;
clc;
seed = 4;
% for p = 1:2
%     subplot(2,1,p);
NodeInfo = csvread('SectorID.csv',1,1);
DiscTime = csvread('Disconnection.csv',1,1);
simTime = 12000;
% Ploting Active node vs. time
titles = ["Static", "Ratio", "Exponential", "Power"];
% LineType = [' -'; '--'; ' :'; '-.'];

    for j = 1:seed
        neighbor = NodeInfo(:,j);
        time_to_die = sort(DiscTime(:,j));
        time = 1:100:simTime;
        ActiveNode = [];
        for i=1:length(time)
            ActiveNode = [ActiveNode 600-length(time_to_die(time_to_die >-
1 & time_to_die < time(i)))];
        end
        % Node per sector
        SectorNode = [];
        for k = 0:8
            SectorNode = [SectorNode length(neighbor(neighbor == k))];
        end

        ActiveSectors = [];
        for m=1:length(time_to_die)
            if(time_to_die(m) > 1 && neighbor(m) >= 0)
                pos = neighbor(m);
                SectorNode(pos+1) = SectorNode(pos+1) - 1;
            end
           ActiveSectors(m) = length(SectorNode(SectorNode > 40));
        end



        switch j
            case 1
                LineType = 'k-';
            case 2
                LineType = 'k--';
            case 3
                LineType = 'k-.';
            case 4
                LineType = 'k:';
```

```
        end

        plot((time_to_die), ActiveSectors, '-');

    %     plot(time, ActiveNode, '-');
        ylim([0 10]);
        xlim([0 simTime]);
        hold on;
    end

    ylabel('Active Sectors');
    xlabel('time (s)');
%     title(sprintf('%ds sampling period', 20*p));

legend('Static','Exponential','Power(10)','Ratio','Location','southwest')
%     legend('seed 1','seed 2','seed 3','seed 4','Location','southwest')
%     legend('CostB 2', 'CostB 5','CostB 10', 'CostB 20');
    grid on;
```

### B.3 Cost function evaluation Matlab script

```
clear all; close all;
E0 = 1000;   % Initial energy
Et = [1000:-1:0];      % Remaining energy
Ni = 1:10;
Hops = 1:6;
linkQ = 0.05:0.01:1;

[x,y] = meshgrid(Et./E0,linkQ);
costFn = (1000*exp(-50*x)).*(1+5*exp(-7*(y.^2)));
subplot(211);
% plot(100*Et/E0, (E0./Et).^3 + 50*(E0./Et).^2);
hold on;
plot(100*Et/E0, (Et/E0).^-10, 'r-');
title('Power cost function (Ei/Ri)^1^0');
ylabel('Cost');

% hold on;
subplot(212)
semilogy(100*Et/E0, 1000*exp(E0./Et));
hold on;
semilogy(100*Et/E0, (Et/E0).^-10, 'r-');
title('Power cost function (Ei/Ri)^5^0');
xlabel('Residual Energy in %');
ylabel('Cost');
% plot(100*Et/E0, 1000*exp(-2*(Et/E0)), 'g-');
```

### B.4 Traffic distribution boxplot Matlab script

```
% This script plots the packet distribution in a 600 node WSN
% It uses the boxplot to highlight the outliers
```

```
close all;
clear all;
% clc;
RNG = 'A1..E5';
packetDistr = csvread('Network Traffic.csv',1,1);
retransmissions = csvread('Network performance - retransmission.csv',1,1);
Retr_per_pkt = sum(retransmissions(:,1:4))./sum(packetDistr(:,1:4));
boxplot(packetDistr, 'outliersize', 2, 'labels', {'Static', 'Exponential',
'Power', 'Ratio'});
% histogram(Retr_per_pkt, [0:3]);
title('Traffic distribution vs cost type');
xlabel('Cost function type');
ylabel('Packets per node');
%set(gca,'ytick',[0:200:1400]);
grid on;
```

### B.5 Ideal deployment visualization script

```
% This script displays nodes deployed in the best case scenario i.e.
evenly
% spaced inside a sector (66 x 66)
clear;
s_range = 10.5;    % sensor range (diameter)
x_lim = 66.6;
y_lim = 66.6;

center_x = s_range/2;


while (center_x < x_lim)
    center_y = s_range/2;
    while (center_y < y_lim)
       viscircles([center_x, center_y], s_range/2, 'EdgeColor', 'b');
       hold on;
       plot(center_x, center_y, 'rx');
       center_y = center_y + s_range;
    end
    hold on;
    center_x = center_x + s_range;
end
hold off;
xlabel('m');
ylabel('m');
xlim([0 x_lim]);
ylim([0 y_lim]);
```

### B.6 Path-loss modeling script

```
clear all; close all;
M = csvread("Book1.csv");
distances = M(:,1);
observed_rssi = M(:,2);
```

```
pathLossExponent = 2.4;
PLd0 = 55;
d0 = 1.0;
sigma = 4.0;
transmissionPower = 0;

pathLossFunc = @(distance) PLd0 + (10 * pathLossExponent *
log10(distance/d0));

plot(distances, observed_rssi,'o')

hold on;
expectedPathLosses = transmissionPower - arrayfun(pathLossFunc,
distances);
plot(distances, expectedPathLosses )
dist = [2 3 4 5 7 9 11 13 15 17 20 21 22 23 24 25 26 27 28 29 30];
Prx_node0 = [-65.65 -69.9 -72.9 -75.2 -78.7 -81.3 -83.42 -85.16 -86.65 -
87.96 -89.65 -90.16 -90.64 -91.1 -91.55 -91.98 -92.39 -92.78 -92.16 -92.52
-92.88];

% plot(dist, Prx_node0, 'gx');
hold off;
ylabel ('Received Power (dB)');
xlabel ('Txer - Rxer distance (m)');
```

### B.7 Packet reception rate modeling script

```
clear all; close all;
Signal = -100:1:-60;
Noise = -100; %dB
PacketLen = 40; % bytes

PRR = (1 - 0.5 * exp(-1*(Signal-Noise)/(2*0.64))).^(8*PacketLen);
Prx_node1 = [-67.1 -71.3 -74.33 -76.65 -80.2 -82.78 -84.87 -86.62 -88.1 -
89.4 -91.1 -91.6 -92.1 -92.56 -92.0 -92.43 -92.84 0 0 0 0];
Prx_node0 = [-65.65 -69.9 -72.9 -75.2 -78.7 -81.3 -83.42 -85.16 -86.65 -
87.96 -89.65 -90.16 -90.64 -91.1 -91.55 -91.98 -92.39 -92.78 -92.16 -92.52
-92.88];
dist = [2 3 4 5 7 9 11 13 15 17 20 21 22 23 24 25 26 27 28 29 30];
PRR_exp_node0 = [100 100 100 100 100 100 100 100 100 100 99 97 90 82 73 59
42 22 11 8 1];
PRR_exp_node1 = [100 100 100 100 100 100 100 100 100 100 92 81 70 41 24 9
1 0 0 0 0];
plot(Signal, 100*PRR);
hold on;
plot(Prx_node0, PRR_exp_node0, 'ro');
axis([-100 -65 0 100]);
xlabel('Received Signal Strength (dB)');
ylabel('Packet Reception rate (%)');
legend( 'Analytical','Emperical');
```

## Appendix C: Castalia network implementation program files

### C.1 Application header file

```
#/**************************************************************************
 #*  Developed at University of Alaska Fairbanks                            *
 #*  Author(s): Osiris Vincent Ntarugera                                    *
 #*  MS Electrical Engineering,                                             *
 #*  Research project - Spring 2017                                         *
 #*  Be sure to reference the author if you use this file (code)            *
 #*                                                                         *
 #**************************************************************************/

#ifndef _TwoLayerNetwork_H_
#define _TwoLayerNetwork_H_

#include "VirtualApplication.h"
#include <vector>

#define CLUSTER_BEACON "Cluster head request to join cluster"
#define FIND_CLUSTER_PACKET_NAME "PA Find cluster head packet"
#define CLUSTER_HEAD_ACK "Request to join cluster accepted packet"
#define NODE_ACK "Ackowledgement to join cluster packet"
#define SPECIAL_NODE "There is a special node in the area"
#define FIND_SPECIAL_NODE "Is there any special node in the area?"
#define AM_SPECIAL_NODE "Yes, I am a special node in the area"
#define OPTIMIZE_NETWORK "Look for neighbors with less hops"
#define OPTIMIZE_NETWORK_ACK "I have fewer hops"
#define FIND_NEIGHBORS_PKT "Find neighbor packet"
#define FIND_NEIGHBORS_ACK_PKT "Reply to find neighbor pkt"
#define DATA_PKT "This is a data packet"
#define FWD_DATA_PKT "Forwarding data pkt"
#define DATA_PKT_ACK "Data packet received"
#define FWD_DATA_PKT_ACK "Forwarding ack"
#define BATTERY_DEAD "My battery is below 5% warning"
#define FIND_NEW_ROUTE "Request new CH packet"
#define FIND_NEW_ROUTE_ACK "Replay to find new CH packet"
#define COST_INFO_PKT "packet contains com. cost information"
#define SET_SECTOR_PKT "Packet contains sector info"

using namespace std;

double LQ_power;              // Link quality power, used to increase the weight of the lQ in the
cost function
                      // Defined gloabally so that it can be used in different classes


struct node_pos {
        double x_coord;
        double y_coord;
};

struct neighbor {             // Class to represent a neighbor node
        int nodeID;
        int dist_to_CH;
        double rssi;

        bool operator==(const neighbor& p) const;
};
struct packet {               // Class to represent a neighbor node
        int pktID;
        int status;
        double dat;
        bool operator==(const packet& p) const;
};
struct timeStamp {            // Class to represent last time a packet was received from node x
        int nodeID;
        double time;
        bool operator==(const timeStamp& t) const;
```

```
};
struct route {
	int routeID;
	int tot_pkt;		//total packets sent on link
	int sx_pkt;		// Successful packets on link
	int length;		// Route length in hops
	double cost;

	bool operator==(const route& r) const;
	bool operator<(const route& s) const;
};
enum PATimers {
	CLUSTER_HEAD_ALERT  = 1,
	FWD_PACKET = 2,
	FIND_CLUSTER_HEAD = 3,
	SPECIAL_NODE_DELAY = 4,
	ENDLESS_TIMER = 5,
	OPTIMIZE_HOPS = 6,
	OPTIMIZE_HOPS_REPLY = 7,
	FIND_SECTOR = 8,
	FIND_NEIGHBORS_ACK = 9,
	TX_DATA_PKT = 10,
	FD_DATA_PKT = 11,
	SPECIAL_NODE_ACK = 12,
	CHECK_BATTERY= 13,
	BUILD_ROUTE_TABLE = 14,
	UPDATE_COST = 15,
	FIND_ROUTE = 16,
	OPTIMIZE_SECTOR = 17,
	FIND_NEIGHBORS = 18,
	SAVE_COMM_COST = 19,
};

class TwoLayerNetwork:public VirtualApplication {
 private:
	int myID;		// node's Network ID
	int myClusterHead;	// node's cluster head ID
	int isPrimary = 0;		// Differentiate cluster heads and members (1: cluster
head, 0: not)

	int foundCluster = 0; // Cluster status, 0 no cluster found, 1 cluster found
	struct node_pos myPosition;	// Node location in (x,y) coordinates
	std::vector<int> myCluster;	// Array of cluster members, only valid for cluster head
	std::vector<route> myRoutes;	// All the possible routes (nodes) a packet can be sent to.
	std::vector<route> badRoutes; // Stores routes with poor link to be avoided in the future
	std::vector<timeStamp> myActiveNode;	// Save the time stamp when a node was last heard
from
	int com_cost_ptk;		// Number of communicaton cost packets sent per node

	int packetSize;		// Data Payload in packets
	int CurrentVersionPckt;		// Version of the current request to join cluster packet
	int requestPcktNum;	// Number of request to join cluster packets received per node
(only cluster head)
	double joinClusterInterval;	// Time delay before requesting to join cluster
	int sample_t;		// Sampling period
	int routeType;		// 0: static, 1: exponential form, 2: power form, 3: simple ratio
form

	double joinClusterIntervalCH; // Time delay before inviting nodes to join cluser (Cluster
head)
	string SpecialNeighbor;		// ID of a node that is having difficulty receiving
pckt ACK
	double batteryLeft;		// How much energy a node is left with (residual energy)
	double consumed_energy;		// Amount of energy consumed to this time
	double E_initial;		// Initial energy
	double lastRadioActivity;	// Last time the radio Txed or Rxed
	bool reply_to_spec_node;	// If true: send ACK to special node, if false: wait
	int hops;			// Number of hops or distance from clusterhead
	double time_to_die;		// Time when node dies
	double disconnect_t;		// Time when node gets disconnected
	float com_cost;			// Communication cost
```

```
        int datapktNum, fdatapktNum;
        int findNeighborAtmpts;
        int specialNodeBeacons;
        int findSecPkt;                     // Packets sent to find sector
        int data_pkt_v;                     // Data packet version
        int prev_pkt;
        int reTx;                   // Number of retransmission
        double myCH_RSSI;           // The rssi of the link from node to CH
        int lowPower;               // 1: battery is low, 0: battery is good
        int myLastCH;               // Save the last CH before disconnection
        int mySectorID;                     // Sector that node belong to
        int connected;              // 0: Node has at least 1 route, 1: node looking for a
route, 2: node completely disconnected
        std::vector<neighbor> myNeighbors;    // Array of neighbors, element represents a
neighbor(ID, dist to sink, and RSSI)

        std::vector<packet> txPackets;              // A list of all packets a node has txed
(whether they came from neighbors or node itself)

        // The following variables are mainly used in the fromNetworkLayer() function, but they
might need to be used outside
        string packetName;
        int pcktNum, sourceaddr;
        double data;
        double cost_varA, cost_varB;         // Parameters in the exponential cost function A*(-
B*Ri/Ei)
        int cost_pow;               // exponent in power form cost function
        //double LQ_thresh;         // link quality threshold

 protected:
        void startup();
        void finishSpecific();
        void fromNetworkLayer(ApplicationPacket *, const char *, double, double);
        void timerFiredCallback(int);
        void update_com_cost();
        void send_data_pkt(string dest, int pktNum, double data, const char* type);
        void send_data_pkt_ack(string destAddr, int pktsource, char* type);
        int updateCH();       // If communication with current CH fails, find a new CH among
neighbors. Return 0 if new CH was found, otherwise return 1
        void updatePktStatus(int pID);              // Update status of a packet when an ACK is
received
        int minNeighborCost();        // Return the lowest cost in the neighbor cost table
        void updateCostTable(int id, double cost);
        void updateLinkQuality(int routeID, int x);
        void routeTableMaintenance();
        void radioPowDraw(int pktSize, int txPow);   // Subtract the appropriate amount of energy
when a packet is sent
        bool isBadRoute(int routeID);               // Remove route from table
        void updateTimeStamp(int id);
};

#endif                          //
```

## C.2 Application .ned file

```
//*************************************************************************
 //*  Developed at University of Alaska Fairbanks                                    *
 //*  Author(s): Osiris Vincent Ntarugera                                  *
 //*  MS Electrical Engineering,                                           *
 //*  Research project - Spring 2017                                  *
 //*  Be sure to reference the author if you use this file (code)          *
 //*                                                                       *
 //*************************************************************************/

package node.application.twoLayerNetwork;

simple TwoLayerNetwork like node.application.iApplication {
```

```
 parameters:
       string applicationID = default ("twoLayerNetwork");
       bool collectTraceInfo = default (false);
       int priority = default(1);
       int routingType = default(0);                          // 0: static, 1: exponential form, 2:
power form, 3: simple ratio form
       int packetHeaderOverhead = default(5);                        // in bytes
       int constantDataPayload =  default(100);             // in bytes
       double joinClusterInterval = default(5);             // in seconds
       double joinClusterIntervalCH = default(1);           // in seconds

       double cost_variableA = default(100);
       double cost_variableB = default(5);
       double cost_power = default(2);
       double linkQ_threshold = default(0);

       bool isSink = default (false);
       int isClusterHead = default(0);              // Differentiate between primary and
secondary nodes
       int batteryLife = default(100);              // Represents the number of packets a node
can send before it dies
       int sampling_period = default(50);    // Node sends a pkt every 50s

                                                    // Primary nodes are also cluster heads
// Add some more parameters specific to this application to be

 gates:
       output toCommunicationModule;
       output toSensorDeviceManager;
       input fromCommunicationModule;
       input fromSensorDeviceManager;
       input fromResourceManager;
}
```

### C.3 Network configuration file

```
#/*****************************************************************************
 #*  Developed at University of Alaska Fairbanks                                  *
 #*  Author(s): Osiris Vincent Ntarugera                                      *
 #*  MS Electrical Engineering,                                               *
 #*  Research project - Spring 2017                                      *
 #*  Be sure to reference the author if you use this file (code)              *
 #*                                                                           *
 #*****************************************************************************/

[General]

include ../Parameters/Castalia.ini

#sim-time-limit = ${simTime=5000s, 10000s, 15000s, 25000s, 30000s, 50000s, 100000s}
sim-time-limit = 6000s

SN.field_x = 200              # meters
SN.field_y = 200              # meters

SN.numNodes = 600

SN.wirelessChannel.onlyStaticNodes = true
#SN.wirelessChannel.sigma = 0
#SN.wirelessChannel.bidirectionalSigma = 0

SN.node[0].xCoor = 33.34
SN.node[0].yCoor = 33.34

SN.node[1].xCoor = 100
SN.node[1].yCoor = 33.34

SN.node[2].xCoor = 166.66
SN.node[2].yCoor = 33.34
```

```
SN.node[3].xCoor = 33.34
SN.node[3].yCoor = 100

SN.node[4].xCoor = 100
SN.node[4].yCoor = 100


SN.node[5].xCoor = 166.66
SN.node[5].yCoor = 100

SN.node[6].xCoor = 33.34
SN.node[6].yCoor = 166.66

SN.node[7].xCoor = 100
SN.node[7].yCoor = 166.66

SN.node[8].xCoor = 166.66
SN.node[8].yCoor = 166.66

SN.deployment = "[9..599]->uniform"

SN.node[*].Communication.Radio.RadioParametersFile = "../Parameters/Radio/CC2420.txt"
SN.node[0..8].Communication.Radio.TxOutputPower = "0dBm"
SN.node[9..599].Communication.Radio.TxOutputPower = "-10dBm"
SN.node[*].Communication.Radio.collisionModel = 2
#SN.node[*].Communication.Radio.mode = "IDEAL"

SN.node[*].ApplicationName = "TwoLayerNetwork"
SN.node[0..8].Application.isSink = true
SN.node[*].Application.sampling_period = 20  #${sampT= 10, 20}
SN.node[*].Application.constantDataPayload = 100
SN.node[0..8].Application.isClusterHead = 1
SN.node[0..8].Application.batteryLife = 10000
SN.node[9..599].Application.batteryLife = 300                        # Initial battery capacity
(mA.sec)
SN.node[9..599].Application.routingType = 2  #${form = 0,1,2,3}                    # Choose
static or non-static routing and cost function type
SN.node[*].Application.cost_variableA = 1000 #${costA= 2, 5}
SN.node[*].Application.cost_variableB = 5    #${costB= 2, 5, 10, 20, 50}
SN.node[*].Application.linkQ_exponent = 1    #${pow = 0, 0.5, 1, 2}
SN.node[*].Application.cost_power = ${pow= 25, 50}

SN.node[*].Application.collectTraceInfo = false

SN.node[*].MobilityManager.collectTraceInfo = false


[Config varyTxPower]
SN.node[*].Communication.Radio.TxOutputPower = ${TxPower="0dBm", "-3dBm", "-7dBm", "-25dBm"}
```

## C.4 Application main file

```
/*****************************************************************************
 *  Developed at University of Alaska Fairbanks                                *
 *  Author(s): Vincent Ntarugera                                               *
 *  MS Electrical Engineering,                                                 *
 *  Research project - Spring 2017
 *  Be sure to reference the author if you use this file (code)                *
 *                                                                             *
 *****************************************************************************/

#include "TwoLayerNetwork.h"
#include <algorithm>
#include <vector>
#include <stdlib.h>                 /* random number generator */
#include <time.h>                   /* time seed for srand */
#include <stdio.h>                  /* printf, NULL */
#include <string>
#include <cmath>
```

```
Define_Module(TwoLayerNetwork);

void TwoLayerNetwork::startup()
{
  myID = atoi(SELF_NETWORK_ADDRESS);
  myClusterHead = -1;
  foundCluster = 0;
  myPosition.x_coord = mobilityModule->getLocation().x;
  myPosition.y_coord = mobilityModule->getLocation().y;
  myCluster.clear();                          // Clear the cluster member array
  packetSize = (int)par("constantDataPayload");
  CurrentVersionPckt = 0;
  requestPcktNum = 0;
  cost_varA = (double)par("cost_variableA");
  cost_varB = (double)par("cost_variableB");
  cost_pow = (int)par("cost_power");
  LQ_power = (double)par("linkQ_threshold");
  isPrimary = (int)par("isClusterHead");
  routeType = (int)par("routingType");
  batteryLeft = ((double)par("batteryLife"));        // Convert mAh to mAs
  E_initial = batteryLeft;
  consumed_energy = 0;
  sample_t = (int)par("sampling_period");
  lastRadioActivity = 0;
  reply_to_spec_node = true;
  hops = 0;
  time_to_die = -1;
  disconnect_t = -1;
  myNeighbors.clear();        // Initialize array to store neighbors
  txPackets.clear();
  com_cost = 0;
  datapktNum = 0;
  fdatapktNum = 0;
  findNeighborAtmpts = 0;
  specialNodeBeacons = 0;
  findSecPkt = 0;
  data_pkt_v = 1;
  prev_pkt = 0;               // Store the last received packet
  reTx = 0;
  myCH_RSSI = 0;
  lowPower = 0;
  mySectorID = -1;
  myRoutes.clear();
  badRoutes.clear();
  com_cost_ptk = 900;
  connected = 0;

  /*-------------------------------------------------------------------------------------------------
------------*/

  srand(myID);
  // joinClusterInterval = (rand()%50);              // Select a random delay < 50s before
sending beacon

  if (isPrimary == 0) //Secondary nodes
  {
    setTimer(UPDATE_COST, 800);
    setTimer(SAVE_COMM_COST, 900);
    setTimer(FIND_CLUSTER_HEAD, rand()%50);
    setTimer(SPECIAL_NODE_DELAY, 150 + rand()% 100);        // set random delay between 100s &
200s to look for unconnected nodes
    setTimer(OPTIMIZE_HOPS, 250 + rand()% 50);                      // Between 150s and 200s
optimize the number of hops
    setTimer(FIND_NEIGHBORS, 500 + rand()% 200);            // Between 320s and 400s nodes send
out pkts to find neighbors
    setTimer(TX_DATA_PKT, 1000 + rand()% 150);                      // Send a packet at a random
time between 450s and 600s
    setTimer(CHECK_BATTERY, 500);                                           // Check battery
status every 10s starting at 500s
    setTimer(OPTIMIZE_SECTOR, 600);                                         // Verify if node is
```

```
the correct sector by comparing with neighbors
  }

  //setTimer(FIND_NEIGHBORS, 310 + rand()% 200);

  if (isPrimary == 1)
    setTimer(FIND_SECTOR, 350 + (myID)*5);
  else
    setTimer(FIND_SECTOR, 400 + rand()%100);

  //declareOutput("Rxed optimize ACK");
  //declareOutput("Requests sent to join cluster");
  //declareOutput("Invitation to join cluster");
  //declareOutput("Rx requests to join cluster");
  // declareOutput("ClusterHead ACK");
  // declareOutput("Special Node ACK");
  // declareOutput("Rxed data pkt");
  declareOutput("Rxed at CH");
  // declareOutput("Forwarded data pkt");
  // declareOutput("Rxed data ACK");
  declareOutput("Find new CH");
  // declareOutput("Packets received");
  // declareOutput("Tx ACK");
  // declareOutput("Sent data pkt");
  declareOutput("Remove route");
  declareOutput("Communication Cost");

}

void TwoLayerNetwork::timerFiredCallback(int timer)
{
  switch (timer) {

    case CLUSTER_HEAD_ALERT:{
      if (foundCluster == 0 && requestPcktNum < 3)                  // This for node who couldn't
reach the hub directly, looking for nodes with link to  hub
      {
        ApplicationPacket* newPckt = createGenericDataPacket(100, CurrentVersionPckt,
packetSize);
        newPckt->setName(CLUSTER_BEACON);
        toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
        radioPowDraw(packetSize, -10);

        setTimer(CLUSTER_HEAD_ALERT, 2);             // Delay 2 sec between packets
        requestPcktNum++;
      }
      break;

    }
    case FIND_CLUSTER_HEAD:{
      if(isPrimary == 0 && foundCluster == 0)
      {
        if (CurrentVersionPckt < 3)           // Try to find a cluster head for 4 times
        {
          ApplicationPacket* newPckt = createGenericDataPacket(100, CurrentVersionPckt,
packetSize);
          newPckt->setName(FIND_CLUSTER_PACKET_NAME);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);

          CurrentVersionPckt++;
          setTimer(FIND_CLUSTER_HEAD, 2);

        }
        else    // If no cluster head is found after 4 attempts, try to find a neighbor with a
cluster head
        {
          setTimer(CLUSTER_HEAD_ALERT, 50);

        }
```

```
      }

      break;
      }
    case SPECIAL_NODE_DELAY:
      {
        if(foundCluster == 0 && isPrimary == 0 && specialNodeBeacons < 4)  // If node was not
able to find a path to hub still
        {
          ApplicationPacket* newPckt = createGenericDataPacket(myID, 10, packetSize);
          newPckt->setName(AM_SPECIAL_NODE);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);
          setTimer(SPECIAL_NODE_DELAY, 2);                                    // Send up to 5
beacons to look for neighbors
          specialNodeBeacons++;
        }
        break;
      }
    case SPECIAL_NODE_ACK:
      {
        ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
        newPckt->setName(CLUSTER_HEAD_ACK);
        toNetworkLayer(newPckt, SpecialNeighbor.c_str());           // Check for any special node
in the surrounding area
        radioPowDraw(packetSize, -10);
        collectOutput("Special Node ACK", stoi(SpecialNeighbor));
      break;
      }
    case OPTIMIZE_HOPS:
      {
        if(hops > 3 || foundCluster == 0)
        {
          ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
          newPckt->setName(OPTIMIZE_NETWORK);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);               // Check for any
special node in the surrounding area
          radioPowDraw(packetSize, -10);
        }

      break;
      }
    case OPTIMIZE_HOPS_REPLY:
      {
        ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
        newPckt->setName(OPTIMIZE_NETWORK_ACK);
        toNetworkLayer(newPckt, SpecialNeighbor.c_str());           // Check for any special node
in the surrounding area
        radioPowDraw(packetSize, -10);
      break;
      }

    case FIND_SECTOR:
      {
        if(isPrimary == 1 )
        {
          if (findSecPkt < 10)
          {
            // toNetworkLayer(createRadioCommand(SET_TX_OUTPUT, -1));
            ApplicationPacket* newPckt = createGenericDataPacket(myID, 1, packetSize);
            newPckt->setName(SET_SECTOR_PKT);
            toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
            findSecPkt++;
            setTimer(FIND_SECTOR, 0.5);
          }
          else
            toNetworkLayer(createRadioCommand(SET_TX_OUTPUT, -10));
        }
        else if(isPrimary == 0 && findSecPkt < 5)
        {
          ApplicationPacket* newPckt = createGenericDataPacket(myCH_RSSI, mySectorID, packetSize);
```

```
        newPckt->setName(SET_SECTOR_PKT);
        toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
        findSecPkt++;
        setTimer(FIND_SECTOR, 2);
      }

    break;
    }

  case OPTIMIZE_SECTOR:
  {
    if(myNeighbors.size() > 0)
    {
      double sec_count = 0;
      int current_sec_count = 0;
      int max_sec = -1;
      double max_sec_count = 0;
      std::vector<neighbor>::iterator n_ptr;
      for(int sec = 0; sec < 9; sec++)
      {
        for(n_ptr = myNeighbors.begin(); n_ptr != myNeighbors.end(); n_ptr++)
        {
          if(n_ptr->dist_to_CH == sec)
            sec_count++;
        }
        if(sec_count > max_sec_count)
        {
          max_sec_count = sec_count;
          max_sec = sec;
        }

        if(sec == mySectorID)
          current_sec_count = sec_count;

        sec_count = 0;
      }


      if(mySectorID == -1 && max_sec_count > 0)
        mySectorID = max_sec;
      else
      {
        if(mySectorID != max_sec && (current_sec_count == 0 || max_sec_count/current_sec_count
>= 3))
          mySectorID = max_sec;
      }
    }

    myNeighbors.clear();            // Empty my neighbors array for other usage
    break;
  }
  case FIND_NEIGHBORS:
  {
    if(findNeighborAtmpts < 4)
    {
      ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
      newPckt->setName(FIND_NEIGHBORS_PKT);
      toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);         // Reply to node looking for
neighbors
      radioPowDraw(packetSize, -10);
      findNeighborAtmpts++;
      setTimer(FIND_NEIGHBORS, 5);
    }
    else if(findNeighborAtmpts < 7)         // Allow nodes with less than 3 routes to continue
searching with a 50s delay
    {
      if(myRoutes.size() <= 2)
      {
        ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
        newPckt->setName(FIND_NEIGHBORS_PKT);
        toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);              // Reply to node
```

```
looking for neighbors
          radioPowDraw(packetSize, -10);
          setTimer(FIND_NEIGHBORS, 50);
          findNeighborAtmpts++;
        }
      }
      break;
    }
    case FIND_NEIGHBORS_ACK:
      {
        ApplicationPacket* newPckt = createGenericDataPacket(com_cost, hops, packetSize);
        newPckt->setName(FIND_NEIGHBORS_ACK_PKT);
        toNetworkLayer(newPckt, SpecialNeighbor.c_str());           // Reply to node looking for
neighbors
        radioPowDraw(packetSize, -10);

      break;
      }
    case TX_DATA_PKT:
    {
      if(connected == 0 && batteryLeft > 0)
      {
        if(datapktNum < 2)
        {
          myClusterHead = minNeighborCost();
          send_data_pkt(to_string(myClusterHead), data_pkt_v*1000 + myID, 1, DATA_PKT);
  // Send random data to next node
          setTimer(TX_DATA_PKT, 0.125);                                    // If no ACK is
received after 3s, resend packet up to 3
          datapktNum++;

          }
        else if (datapktNum < 4)
        {

          updateCH();
          myClusterHead = minNeighborCost();
          send_data_pkt(to_string(myClusterHead), data_pkt_v*1000 + myID, 1, DATA_PKT);
  // Send random data to next node
          setTimer(TX_DATA_PKT, 0.125);                                    // If no ACK is
received after 3s, resend packet up to 3
          datapktNum++;
        }
        else
        {
          datapktNum = 0;
          data_pkt_v++;                                        // Update packet version
          setTimer(TX_DATA_PKT, sample_t);
        }
      }
      break;
    }
    case FD_DATA_PKT:
    {
      if (fdatapktNum < 2)
      {
        myClusterHead = minNeighborCost();
        send_data_pkt(to_string(myClusterHead), pcktNum, data+1, FWD_DATA_PKT);
        setTimer(FD_DATA_PKT, 0.125);
        fdatapktNum++;
      }
      else if(fdatapktNum < 4)
      {
        updateCH();
        myClusterHead = minNeighborCost();
        send_data_pkt(to_string(myClusterHead), pcktNum, data+1, FWD_DATA_PKT);
        setTimer(FD_DATA_PKT, 0.125);
        fdatapktNum++;
      }
      else
        fdatapktNum = 0;
```

```
      break;
    }
    case CHECK_BATTERY:
    {
      if(batteryLeft > 0 && batteryLeft/E_initial < 0.05)          // If battery is below 5%,
send a warning to all neighbors
      {
        ApplicationPacket* newPckt = createGenericDataPacket(0, 10, packetSize);
        newPckt->setName(BATTERY_DEAD);
        toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
        radioPowDraw(packetSize, -10);
        lowPower = 1;
      }
      else
        setTimer(CHECK_BATTERY, 10);

      break;
    }

    case BUILD_ROUTE_TABLE:
    {

      if(batteryLeft/E_initial > 0.1 && connected == 0)
      {
        ApplicationPacket* newPckt = createGenericDataPacket(com_cost, hops, packetSize);
        newPckt->setName(FIND_NEW_ROUTE_ACK);
        toNetworkLayer(newPckt, SpecialNeighbor.c_str());
        radioPowDraw(packetSize, -10);
      }

      break;
    }
    case UPDATE_COST:
    {
      if(batteryLeft > 0 && routeType != 0)
      {
        update_com_cost();
        routeTableMaintenance();                    // Remove routes with poor link Q, and look
for more route if necessary
        setTimer(UPDATE_COST, 10);                  // Update cost every 10s
      }
      break;
    }
    case SAVE_COMM_COST:              // Save the communication cost every 100s
    {
      // if(myID == 60){
      //  collectOutput("Communication Cost", floor(100.0*batteryLeft/E_initial), "", com_cost);
      //  com_cost_ptk += 100;
      //  setTimer(SAVE_COMM_COST, 50);
      // }

      break;
    }
    case FIND_ROUTE:
    {
      if(batteryLeft/E_initial > 0.1)
      {
        if(findNeighborAtmpts < 2)
        {
          ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
          newPckt->setName(FIND_NEW_ROUTE);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);

          findNeighborAtmpts++;
          setTimer(FIND_ROUTE, 5);
        }
        else if(findNeighborAtmpts < 5 && myRoutes.size() == 0)
        {

          ApplicationPacket* newPckt = createGenericDataPacket(hops, 10, packetSize);
```

```
          newPckt->setName(FIND_NEW_ROUTE);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);

          findNeighborAtmpts++;
          setTimer(FIND_ROUTE, 10);

        }
        else  // Node was unable to find routes to hub, let neighbors know
        {
          connected = 2;
          ApplicationPacket* newPckt = createGenericDataPacket(0, 10, packetSize);
          newPckt->setName(BATTERY_DEAD);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);
        }
      }
      break;
    }
  }
}

void TwoLayerNetwork::fromNetworkLayer(ApplicationPacket *rcvPacket, const char *source, double
rssi, double lqi)
{
  if(batteryLeft > 5)
  {


    packetName = rcvPacket->getName();
    pcktNum = rcvPacket->getSequenceNumber();
    sourceaddr = atoi(source);
    data = rcvPacket->getData();

      if(packetName.compare(CLUSTER_BEACON) == 0)
      {
        if (isPrimary == 1 || foundCluster == 1)     // If node is not a hub but it has found a
path to the hub, send ACK
        {
          SpecialNeighbor = source;
          setTimer(SPECIAL_NODE_ACK, rand() % 5);
        }

      }
      else if(packetName.compare(FIND_CLUSTER_PACKET_NAME) == 0)
      {
        if((isPrimary == 1))          // Respond to nodes looking for path to the hub
        {
          //myCluster.push_back(sourceaddr);                // Add node ID to the list of cluster
members
          ApplicationPacket* newPckt = createGenericDataPacket(hops, 1, packetSize);       // Send
ACK back to node, to acknowledge the request to join cluster
          newPckt->setName(CLUSTER_HEAD_ACK);
          toNetworkLayer(newPckt, source);
          radioPowDraw(packetSize, -10);

        }
      }
      else if(packetName.compare(CLUSTER_HEAD_ACK) == 0)
      {


        if ((foundCluster == 0) || hops > ((int)data + 1))   // If node has not found path to hub
or current path is longer than this one
        {
          myClusterHead = sourceaddr;         // Update cluster head ID
          foundCluster = 1;                           // Update cluster membership status
          hops = (int)data + 1;                       // Update distance to CH or hops
          myCH_RSSI = rssi;
```

```
      route r({sourceaddr, 1, 1, pcktNum, data});
      std::vector<route>::iterator ptr = std::find(myRoutes.begin(), myRoutes.end(), r);
      if(ptr == myRoutes.end())            // If route doesn't exist in table, add it
        myRoutes.push_back(r);

        cancelTimer(FIND_CLUSTER_HEAD);                    // Cancel timer to search for path
      }
    }
    else if(packetName.compare(AM_SPECIAL_NODE) == 0)
    {
      if(foundCluster == 1 || isPrimary == 1)
      {
        SpecialNeighbor = source;
        setTimer(SPECIAL_NODE_ACK, rand()%10);
      }

    }
    else if(packetName.compare(OPTIMIZE_NETWORK) == 0)
    {
      if(hops < (int)data -1)
      {

        SpecialNeighbor = source;
        setTimer(OPTIMIZE_HOPS_REPLY, rand()%10);

      }
    }
    else if(packetName.compare(OPTIMIZE_NETWORK_ACK) == 0)
    {
      myClusterHead = sourceaddr;          // Update cluster head ID
      foundCluster = 1;                    // Update cluster membership status
      hops = (int)data + 1;
      myCH_RSSI = rssi;


    }
    else if(packetName.compare(FIND_NEIGHBORS_PKT) == 0)
    {

      SpecialNeighbor = source;
      setTimer(FIND_NEIGHBORS_ACK, rand()%20);            // Send find_neighbor ack after x
secs < 10s

    }
    else if(packetName.compare(FIND_NEIGHBORS_ACK_PKT) == 0)
    {

      if (isPrimary == 0 && hops == 0)
      {
        route r({sourceaddr, 1, 1, pcktNum, data});
        std::vector<route>::iterator ptr = std::find(myRoutes.begin(), myRoutes.end(), r);
        if(ptr == myRoutes.end())          // If route doesn't exist in table, add it
          myRoutes.push_back(r);

        myClusterHead = sourceaddr;        // Update cluster head ID
        foundCluster = 1;                  // Update cluster membership status
        hops = pcktNum + 1;
        myCH_RSSI = rssi;


      }

        route r({sourceaddr, 1, 1, pcktNum, data});
        std::vector<route>::iterator ptr = std::find(myRoutes.begin(), myRoutes.end(), r);
        if(ptr == myRoutes.end())          // If route doesn't exist in table, add it
          myRoutes.push_back(r);

    }
    else if(packetName.compare(DATA_PKT) == 0)
    {
      packet pk({pcktNum, 1, data});
```

```
        if(std::find(txPackets.begin(), txPackets.end(), pk) != txPackets.end())   // If packet
has been received before, send 3 ACK
        {
          send_data_pkt_ack(to_string(sourceaddr), pcktNum, DATA_PKT_ACK);
          reTx++;
        }
        else
        {
          send_data_pkt_ack(to_string(sourceaddr), pcktNum, DATA_PKT_ACK);         // Send ACK
with the source addr when data pkt is rxed
          // collectOutput("Rxed data pkt", sourceaddr);

          if(isPrimary == 0)          // Forward packet if you are not a primary node
          {
            myClusterHead = minNeighborCost();
            send_data_pkt(to_string(myClusterHead), pcktNum, data+1, FWD_DATA_PKT);
            //collectOutput("Forwarded data pkt", sourceaddr);
            setTimer(FD_DATA_PKT, 0.125);            // Resend pkt after 1 secs if no ack is rxed
            fdatapktNum++;

          }
          else if(isPrimary == 1)
          {
            packet p({pcktNum, 1, data});            // Add new packet to array if not already
there
            if(std::find(txPackets.begin(), txPackets.end(), p) == txPackets.end())        //
Check if packet exists in array
              txPackets.push_back(p);

            updateTimeStamp(pcktNum%1000);           // Update packet reception time stamp
          }
        }

        if(std::find(myCluster.begin(), myCluster.end(), sourceaddr) == myCluster.end())
  // Check if source node exists in array
          myCluster.push_back(sourceaddr);
      }
      else if(packetName.compare(FWD_DATA_PKT) == 0)
      {
        packet pk({pcktNum, 1, data});
        if(std::find(txPackets.begin(), txPackets.end(), pk) != txPackets.end())   // If packet
has been received before, send 3 ACK
        {
          send_data_pkt_ack(to_string(sourceaddr), pcktNum, FWD_DATA_PKT_ACK);
          reTx++;
        }
        else
        {
          send_data_pkt_ack(to_string(sourceaddr), pcktNum, FWD_DATA_PKT_ACK);            // Send
ACK with the source addr when data pkt is rxed


          if(isPrimary == 0)          // Forward packet if you are not a primary node
          {
            myClusterHead = minNeighborCost();
            send_data_pkt(to_string(myClusterHead), pcktNum, data+1, FWD_DATA_PKT);
            setTimer(FD_DATA_PKT, 0.125);            // Resend pkt after 1 secs if no ack is rxed
            fdatapktNum++;

          }
          else if(isPrimary == 1)
          {
            packet p({pcktNum, 1, data});            // Add new packet to array if not already
there
            if(std::find(txPackets.begin(), txPackets.end(), p) == txPackets.end())        //
Check if packet exists in array
              txPackets.push_back(p);

            updateTimeStamp(pcktNum%1000);                  // Update packet reception time stamp
          }
        }
```

```
        if(std::find(myCluster.begin(), myCluster.end(), sourceaddr) == myCluster.end())
  // Check if source node exists in array
          myCluster.push_back(sourceaddr);
      }
      else if(packetName.compare(DATA_PKT_ACK) == 0)
      {

        updatePktStatus(pcktNum);
        updateCostTable(sourceaddr, data);
        updateLinkQuality(sourceaddr, 1);
        cancelTimer(TX_DATA_PKT);                    // Cancel timer to re-send the packet, if at
least one pkt has been txed
        datapktNum = 0;                                          // reset packet counter
        data_pkt_v++;                                // Update packet version
        setTimer(TX_DATA_PKT, sample_t);             // Start a timer upon receiving ACK
to send a new packet
      }
      else if(packetName.compare(FWD_DATA_PKT_ACK) == 0)
      {

        cancelTimer(FD_DATA_PKT);
        fdatapktNum = 0;
        updatePktStatus(pcktNum);
        updateCostTable(sourceaddr, data);
        updateLinkQuality(sourceaddr, 1);
      }
      else if(packetName.compare(BATTERY_DEAD) == 0)
      {
        if(myClusterHead == sourceaddr)
        {
          disconnect_t = SIMTIME_DBL(getClock());            // Capture time when node gets
disconnected
          // myClusterHead = -1;
          ApplicationPacket* newPckt = createGenericDataPacket(0, 10, packetSize);
          newPckt->setName(BATTERY_DEAD);
          toNetworkLayer(newPckt, BROADCAST_NETWORK_ADDRESS);
          radioPowDraw(packetSize, -10);
        }


        route rt({sourceaddr, 1, 1, 1, data});
        std::vector<route>::iterator it = std::find(myRoutes.begin(), myRoutes.end(), rt);
        if(it != myRoutes.end())
        {
          myRoutes.erase(it);
          badRoutes.push_back(rt);
        }

      }
      else if(packetName.compare(FIND_NEW_ROUTE) == 0)
      {
        if(data > hops)                        // Reply to find_new_route packet if I am closer to
the hub
        {
          SpecialNeighbor = source;
          setTimer(BUILD_ROUTE_TABLE, rand()%5);
        }

      }
      else if(packetName.compare(FIND_NEW_ROUTE_ACK) == 0)
      {
        if(!isBadRoute(sourceaddr))                  // Check if route is not among bad routes
        {
          route r({sourceaddr, 1, 1, pcktNum, data});
          if(myRoutes.size() ==0)
          {
            myRoutes.push_back(r);                           // add new route to table
            connected = 0;
            if(getTimer(TX_DATA_PKT) == 0)           // Restart data packet timer if it is not
active
```

```
            setTimer(TX_DATA_PKT, sample_t);

          collectOutput("Find new CH", sourceaddr);
        }
        else
        {
          if(find(myRoutes.begin(), myRoutes.end(), r) == myRoutes.end())
          {
            myRoutes.push_back(r);                          // add new route to table
            connected = 0;
            if(getTimer(TX_DATA_PKT) == 0)          // Restart data packet timer if it is not
active
              setTimer(TX_DATA_PKT, sample_t);

            collectOutput("Find new CH", sourceaddr);
          }
        }
      }
    }
    else if(packetName.compare(COST_INFO_PKT) == 0)
    {
      if(isPrimary == 0 && pcktNum < 10) //&& (pcktNum/100) == mySectorID)
      {
        route r({sourceaddr, 1, 1, pcktNum, data});
        std::vector<route>::iterator ptr = std::find(myRoutes.begin(), myRoutes.end(), r);
        if(ptr == myRoutes.end())           // If route doesn't exist in table, add it
          myRoutes.push_back(r);
        else
        {

          if(ptr->length > pcktNum)
          {                           // Otherwise, update cost
            ptr->cost = data;
            ptr->length = pcktNum;
          }
        }

        setTimer(BUILD_ROUTE_TABLE, rand()%5);
      }
    }
    else if(packetName.compare(SET_SECTOR_PKT) == 0)
    {
      if(sourceaddr < 9)
      {

        if(mySectorID == -1)
        {
          mySectorID = data;

          myCH_RSSI = rssi;

        }
        else
        {
          if(std::abs(myCH_RSSI) > std::abs(rssi))
          {
            mySectorID = data;
            myCH_RSSI = rssi;
          }
        }
        // setTimer(FIND_SECTOR, rand()%10);
      }
      else
      {

        neighbor n({sourceaddr, pcktNum, data});           // Add neighbor to list
        if(std::find(myNeighbors.begin(), myNeighbors.end(), n) == myNeighbors.end())
          myNeighbors.push_back(n);


      }
```

```
        trace() << "Received sector Information packet.";
      }
      else
        trace() << "Unkown packet type from node " << sourceaddr;

      radioPowDraw(packetSize, 0);
      // Update residual energy upon receiving a packet
      if(batteryLeft == 0)
        time_to_die = SIMTIME_DBL(getClock());

      collectOutput("Packets received");
  }
}


void TwoLayerNetwork::finishSpecific()
{
  if(isPrimary == 0)
  {
     // declareOutput("My Cluster head");                    // Shows which nodes have cluster
heads(belong to a cluster)
     // collectOutput("My Cluster head", "", myClusterHead);

     declareOutput("Retransmissions");                       // Shows which nodes have cluster
heads(belong to a cluster)
     collectOutput("Retransmissions", "", reTx);

    declareOutput("Sector ID");
    collectOutput("Sector ID", "", mySectorID);

    // declareOutput("Neighbors");
    // // collectOutput("Neighbors", "", myRoutes.size());
    // for (int i = 0; i < myRoutes.size(); i++)
    // {
    //  collectOutput("Neighbors", myRoutes.at(i).routeID, "",
(double)myRoutes.at(i).sx_pkt/myRoutes.at(i).tot_pkt);
    // }
    // declareOutput("Neighbor count");
    // collectOutput("Neighbor count", "", myRoutes.size());

    declareOutput("Data traffic");
    collectOutput("Data traffic", "", txPackets.size());
    // collectOutput("Data traffic", "", txPackets.size());
    // int sxful_pkt = 0;
    // for (int i = 0; i < txPackets.size(); i++)
    // {
    //  if(txPackets.at(i).status == 0)                      // Packet was not txed successfully
    //    sxful_pkt++;
    // }
    // if(txPackets.size() > 0)
    //  collectOutput("Data traffic", "", (double)sxful_pkt/txPackets.size());
    // else
    //  collectOutput("Data traffic", "", 0);


  }
  else
  {

    collectOutput("Rxed at CH", "", txPackets.size());             // Collect all packets
received at CHs

    declareOutput("Packet distance");
    int dist = 0;
    for(int k = 0; k < txPackets.size(); k++)
    {
      dist += txPackets.at(k).dat;
    }
    collectOutput("Packet distance", "", (double)dist/txPackets.size());
    declareOutput("Disconnection time");
```

```
    for(int j = 0; j < myActiveNode.size(); j++)
    {
      collectOutput("Disconnection time", myActiveNode.at(j).nodeID, "",
myActiveNode.at(j).time);
    }

  }


    // declareOutput("Battery Status");
    // collectOutput("Battery Status", "", batteryLeft);

    // declareOutput("Consumed Energy ratio");
    // collectOutput("Consumed Energy ratio", "", 100*consumed_energy/(E_initial - batteryLeft));

    // update_com_cost();
    // declareOutput("Communication Cost");
    // collectOutput("Communication Cost", "", hops);

    declareOutput("Packet version");
    collectOutput("Packet version", "", data_pkt_v-1);

    declareOutput("Dependents");
    collectOutput("Dependents", "", myCluster.size());

    declareOutput("Time to die");
    collectOutput("Time to die", "", time_to_die);

}


// Definition of the == operator for class neighbor
bool neighbor::operator==(const neighbor& p) const
{
  return this->nodeID == p.nodeID;
}

// Definition of the == operator for class neighbor
bool packet::operator==(const packet& p) const
{
  return this->pktID == p.pktID;
}

bool route::operator==(const route& r) const
{
  return this->routeID == r.routeID;
}
bool route::operator<(const route& s) const
{
  //return this->cost*(1 + 5*exp(-7*pow((double)this->sx_pkt/this->tot_pkt, 2))) < s.cost * (1 +
5*exp(-7*pow((double)s.sx_pkt/s.tot_pkt, 2)));
  // return this->cost*pow((double)this->tot_pkt/this->sx_pkt, LQ_power) < s.cost *
pow((double)s.tot_pkt/s.sx_pkt, LQ_power);
  return this->cost < s.cost;
}
bool timeStamp::operator==(const timeStamp& t) const
{
  return this->nodeID == t.nodeID;
}
void TwoLayerNetwork::update_com_cost(){

  if(batteryLeft > 0 && myRoutes.size() > 0)
  {
    minNeighborCost();
    route minRoute = myRoutes.at(0);
    switch(routeType)
    {
      case 1:
      {
        com_cost = cost_varA*exp(-cost_varB*batteryLeft/E_initial);
        break;
```

```
      }
      case 2:
      {
        com_cost = pow(E_initial/batteryLeft, cost_pow);
        break;
      }
      case 3:
      {
        com_cost = (E_initial/batteryLeft);
        break;
      }
    }

    com_cost += minRoute.cost;//*pow((double)minRoute.tot_pkt/minRoute.sx_pkt, LQ_power);

  }

}

void TwoLayerNetwork::send_data_pkt(string destAddr, int pktVersion, double d, const char* type){

  if(batteryLeft > 0)
  {

    if(stoi(destAddr) >= 0)
    {
      ApplicationPacket* newPckt = createGenericDataPacket(d, pktVersion, packetSize);
      newPckt->setName(type);                                          // Differenciate
between sending vs forwarding data
      toNetworkLayer(newPckt, destAddr.c_str());            // Send data packet to destination
node
      radioPowDraw(packetSize, -10);
      if(batteryLeft == 0)
        time_to_die = SIMTIME_DBL(getClock());
      // Add new data packet to the array, with status 1 meaning no ACK
      packet p({pktVersion, 1, d});
      if(std::find(txPackets.begin(), txPackets.end(), p) == txPackets.end())             //
Check if packet exists in array
        txPackets.push_back(p);
      else
        reTx++;                                          // This packet is being retransmitted
      updateLinkQuality(stoi(destAddr), 0);          // Update the quality for this link (0:
incremement total packets)


    }
  }
}

void TwoLayerNetwork::send_data_pkt_ack(string destAddr, int pktsource, char* type){

  if(batteryLeft > 0)
  {
    if(stoi(destAddr) >= 0)
    {
      ApplicationPacket* newPckt = createGenericDataPacket(com_cost, pktsource, packetSize);
      newPckt->setName(type);
      toNetworkLayer(newPckt, destAddr.c_str());            // Send data packet to destination
node
      radioPowDraw(11, -10);                 // ACK packet is 11 bytes long
      if(batteryLeft == 0)
        time_to_die = SIMTIME_DBL(getClock());
    }
  }


}
int TwoLayerNetwork::updateCH(){                                                  // Run thru my
neighbors looking for another node at the same distance as my current CH, if not one hop long etc

  if(myRoutes.size() > 0)
```

```
  {
    minNeighborCost();
    return 1;
  }
  else
    return 0;
}

void TwoLayerNetwork::updatePktStatus(int pID){

  std::vector<packet>::iterator it;
  packet p({pID, 0, 0});
  it = find(txPackets.begin(), txPackets.end(), p);
  if(it != txPackets.end())
    it->status = 0;
}

int TwoLayerNetwork::minNeighborCost(){

  if(routeType == 0)                  // Static routing, keep clusterhead unchanged
    return myClusterHead;
  else
  {
    if(myRoutes.size()==0)
    {
      return -1;
    }
    else
    {
      std::sort(myRoutes.begin(), myRoutes.end());
      return myRoutes.at(0).routeID;
    }
  }
}

void TwoLayerNetwork::updateCostTable(int id, double cost){

  std::vector<route>::iterator r_ptr;
  route r({id, 1, 1, 1, cost});
  r_ptr = find(myRoutes.begin(), myRoutes.end(), r);
  if(r_ptr != myRoutes.end())
    r_ptr->cost = cost;
  else
    myRoutes.push_back(r);
}

void TwoLayerNetwork::updateLinkQuality(int routeID, int x){

  std::vector<route>::iterator r_ptr;
  route r({routeID, 1, 1, 1, 0});
  r_ptr = find(myRoutes.begin(), myRoutes.end(), r);
  if(r_ptr == myRoutes.end())                // Route doesn't exist in my route table
    return;
  if(x == 0)
    r_ptr->tot_pkt++;
  else if(x==1)
    r_ptr->sx_pkt++;
}

void TwoLayerNetwork::routeTableMaintenance(){

  std::vector<route>::iterator rt;
  if(myRoutes.size() > 0)
  {
    for (rt = myRoutes.begin(); rt < myRoutes.end(); rt++)
    {
      if((double)rt->sx_pkt/rt->tot_pkt < 0.2)              // If route's link Q is < LQ
Threshold
      {
        myRoutes.erase(rt);
        badRoutes.push_back(*rt);
```

```
        collectOutput("Remove route", rt->routeID);
      }
    }
  }
  else if(myRoutes.size() == 0 && connected < 2)
  {

    connected = 1;                    // No routes available
    findNeighborAtmpts = 0;
    setTimer(FIND_ROUTE, 2);
    cancelTimer(TX_DATA_PKT);

  }
}

void TwoLayerNetwork::radioPowDraw(int pktSize, int txPow){

  if(batteryLeft > 0)
  {
    double txCurrent;
    int dataRate = 250000;            // 250Kbps
    switch(txPow)
    {
      case 0:
      {
        txCurrent = 17.4;             //mA
        break;
      }
      case -3:
      {
        txCurrent = 15.4;             //mA
        break;
      }
      case -5:
      {
        txCurrent = 14;               //mA
        break;
      }
      case -10:
      {
        txCurrent = 11;               //mA
        break;
      }
      case -15:
      {
        txCurrent = 9.9;              //mA
        break;
      }
      case -25:
      {
        txCurrent = 8.5;              //mA
        break;
      }
    }

    consumed_energy += txCurrent * ((double)pktSize*8/dataRate); // Energy consumed by the radio
+ (SIMTIME_DBL(getClock()) - lastRadioActivity) * (0.02 + 0.002 + 0.006);
    batteryLeft = batteryLeft - txCurrent * ((double)pktSize*8/dataRate);
    batteryLeft = batteryLeft - (SIMTIME_DBL(getClock()) - lastRadioActivity) * (0.02 + 0.002 +
0.006);  // Assume 20µA Main radio, 2µA MCU, 6µA WuRx
    lastRadioActivity = SIMTIME_DBL(getClock());

    if(batteryLeft < 0)
      time_to_die = SIMTIME_DBL(getClock());

}
}

bool TwoLayerNetwork::isBadRoute(int routeID)              // Return true if this is a poor link
route
{
```

```
  std::vector<route>::iterator r_ptr;
  route r({routeID, 1, 1, 1, 0});
  r_ptr = find(myRoutes.begin(), myRoutes.end(), r);

  if(r_ptr == myRoutes.end())
    return false;
  else
    return true;

}

void TwoLayerNetwork::updateTimeStamp(int id)
{
  timeStamp t({id, SIMTIME_DBL(getClock())});
  std::vector<timeStamp>::iterator t_ptr = find(myActiveNode.begin(), myActiveNode.end(), t);

  if(t_ptr == myActiveNode.end())
    myActiveNode.push_back(t);
  else
    t_ptr->time = SIMTIME_DBL(getClock());
}
```